DOCTOR OF SCIENCES

Feature-based configuration: collaborative, dependable, and controlled

Hubaux, Arnaud

Award date:
2012

Awarding institution:
University of Namur

PReCISE Research Center
University of Namur
Faculty of Computer Science
Belgium

Feature-based Configuration: Collaborative, Dependable, and Controlled

Arnaud Hubaux

Thèse présentée en vue de l'obtention du grade de Docteur en Sciences

PhD Thesis January 2012

Rempart de la Vierge, 13
B - 5000 Namur (Belgique)

Imprimé en Belgique

ISBN : 978-2-87037 -753-6

Dépôt légal: D / 2012 / 1881 / 6

Jury

Prof. Jean-Noël Colin, University of Namur, Belgium
Prof. Krzysztof Czarnecki, University of Waterloo, Canada
Prof. Naji Habra, University of Namur, Belgium (chair)
Prof. Patrick Heymans, University of Namur, Belgium (advisor)
Prof. Kim Mens, Catholic University of Louvain, Belgium
Prof. Pierre-Yves Schobbens, University of Namur, Belgium (co-advisor)
Prof. Laurent Schumacher, University of Namur, Belgium

Abstract

A growing share of products expose sophisticated *configurability* to handle variations in user and context requirements. Configurators usually rely on variability models, like *feature models* (FMs), that structure and constrain the available options. However, most feature-based configuration techniques do not scale well to collaborative contexts. They offer limited mechanisms to determine responsibilities and access rights, to schedule configuration tasks, and to resolve conflicts. Fragments of solutions already exist but a unified and formal foundation for collaborative feature-based configuration is still missing.

To provide enhanced control and guidance, we specify responsibilities and rights with views on the FM. Views establish insulated spaces in which users can safely configure the part of the FM assigned to them. The configuration of these views is regulated by a workflow that defines the configuration process. The result of that combination is a new formalism called *feature-based configuration workflow*. Finally, to handle conflicts between user decisions, we develop a *range fix* generation algorithm. These concepts and their properties are integrated into a sound mathematical framework.

Our contribution is motivated and illustrated through several real-world applications: a product line of meeting management applications (PloneMeeting), a product line of communication protocols used in the aerospace industry (CFDP), the Linux kernel, and an operating system for embedded applications (eCos). The definitions and algorithms are implemented in a toolset that extends SPLOT, an open source configuration environment, and YAWL, a comprehensive workflow management environment. This toolset demonstrates the efficiency and applicability of our contribution.

Résumé

Un nombre croissant de produits offrent des mécanismes de *configurarion* sophistiqués pour gérer les variations entre les exigences des utilisateurs et du contexte. Habituellement, les configurateurs reposent sur des modèles de variabilité, tels que les *feature models* (FMs), qui structurent et contraignent les options disponibles. Cependant, la plupart des techniques de configuration basées sur les FMs s'appliquent difficilement aux contexts collaboratifs. Ils offrent des mécanismes limités pour déterminer les responsabilités et droits d'accès, organiser des tâches de configuration, et résoudre des conflits. Des fragments de solutions existent mais une fondation formelle et unifiée pour la configuration collaborative basée sur les FMs est manquante.

Pour fournir un contrôle et une guidance améliorés, nous spécifions les responsabilités et droits d'accès avec des vues sur le FM. Les vues établissent des espaces isolés dans lesquels les utilisateurs configurent la partie du FM qui leur est assignée. La configuration de ces vues est regulée par un workflow qui définit le processus de configuration. Le résultat de cette combinaison est un nouveau formalisme appelé *feature-based configuration workflow*. Finalement, pour traiter les conflits entre les décisions des utilisateurs, nous développons un algorithme de génération de *range fixes*. Ces concepts et leurs propriétés sont intégrés dans un framework mathématique cohérent.

Notre contribution est motivée et illustrée par plusieurs applications : une ligne de produits d'applications de gestions de réunions (PloneMeeting), une ligne de produit de protocoles de communications utilisée dans l'industrie aéronautique (CFDP), le kernel Linux, et un système d'exploitation pour applications embarquées (eCos). Les définitions et algorithmes sont implémentés dans un toolset qui étend SPLOT, un environnement de configuration open source, et YAWL, un système complet de gestion de workflow. Ce toolset démontre l'efficacité et l'applicabilité de notre contribution.

Acknowledgements

This whole adventure started on the impulse of two men: Prof. Pierre-Yves Schobbens and Prof. Patrick Heymans. Pierre-Yves supervised my Master's thesis and enrolled me in the MoVES project. His vast knowledge and razor-sharp rigour repeatedly helped me improve and simplify formal definitions and algorithms.

Over four years, Patrick put some many caps on that I barely remember them all: supervisor, mentor, overly picky reviewer, friend, room mate, host, mad neighbour... His sniper's eye spots the weaknesses of any piece of text and his magic hand rephrases it frustratingly plainly. He markedly improved all my papers and taught me a lot about paper writing.

The MoVES project has been a cradle for the new-born researcher I was. The three work packages that took me on board (WP2, 3, and 4) taught me to think, act, and communicate as a researcher. The collaborations built within the network resulted in several joint publications integrated into this manuscript.

As a student and researcher, I had the chance to work with Andreas Classen for over nine years. His scientific skills, discipline, dedication, and outspokenness make working with him both a pleasure and a challenge. Outside the office, he is a hell of a party-goer, always up for a beer, and eager to debate. Several chapters of this manuscript bear his indelible mark.

All my co-authors own a significant piece of this manuscript, too. I also thank the industrial partners who helped me come back to reality and fed me with valuable insight at conferences, meetings, or boarding areas. I am particularly thankful to Jean-Marc Astesana, Danilo Beuche, Gaëtan Delannay, and Herman Hartmann.

For more than three years, I shared an office with a bunch of fantastic guys who all contributed to the ideas presented here: Raimundas, Germain, Quentin, Jean-Christophe, Raphaël, and Ebrahim. Special thanks to Ebrahim who spared no effort working on the toolset despite the difference in timezone, and Anthony Cleve whose acute understanding of the research game kept us debating for hours.

The stay at the GSD lab at the University of Waterloo (ON, Canada) was a milestone in my short research career. I thank Prof. Krzysztof Czarnecki for welcoming me in his lab for one year. Despite his busy schedule, he always managed to find some time to discuss and put drifting ideas back on track. His passion and vision were genuinely inspiring. I also thank the GSD lab for the lively discussions, and Yingfei and Steven for the nights they spend wrapping up our work on fix generation. I also want to express my gratitude to Marcilio for his help with SPLOT.

This stay abroad also put an heavy load of red tape on my parents during my absence. I thank them for their patience. I also thank my brother who kindly accepted to proofread several parts of the manuscript and give his outsider's feedback.

Finally and foremost, I am immensely grateful to my fiancée, Marie-Charlotte Druet, who kept me going through the dire straits of this thesis. Without the slightest hesitation, she followed me in Toronto for one year, leaving her family and friends behind. I apologize for all the rough patches she went through and dedicate this manuscript entirely to her.

This work is sponsored by the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy, under the MoVES project, and IN.WALLONIA-Brussels International under the (IN.WBI) Excellence grants.

Contents

Glossary

FBC	Feature-based Configuration
FCW	Feature-based Configuration Workflow
FM	Feature Model
MLSC	Multi-Level Staged Configuration
SCM	Software Configuration Management
SD	Strong Dependency Set
SoC	Separation of Concerns
SPL	Software Product Line
SPLE	Software Product Line Engineering
WD	Weak Dependency Set
YAWL	Yet Another Workflow Language

Chapter

1 Introduction

1.1 *"To Foresee Is To Rule"*

Born in France, Blaise Pascal (1623–1662) pioneered work in mathematics, physics and philosophy, and invented the first mechanical calculator called the *Pascaline*. Over 300 years ago, he captured the essence of this thesis in five words: *"To foresee is to rule"*. Traced back to software, foresight into what decisions should be left in the hand of customers is the cornerstone to ideally tailored applications. But how do *tailor-made* and *general-purpose* software differ?

Geared towards the mass market, *general-purpose* applications like web browsers, PDF readers, and email clients must satisfy most customer requirements out of the box. More specialised applications like compilers, CASE tools, and professional photo editing applications fall under the same umbrella: They provide one-size-fits-all solutions with little room for user preferences.

In contrast, *tailor-made* software aims at accommodating customer preferences prior to delivery or use. The glaring difference with general-purpose software is that customers are presented with a set of options rather than an exhaustive list of software products. That intentional specification of the product portfolio drastically expands the range of variants offered to customers. By progressively deciding which option to retain or discard, customers narrow down their selection to one product. This process, commonly called *configuration*, has been embraced by a wide range of domains such as operating systems, healthcare equipment, semiconductors, and automotive.

To gain that competitive edge, dedicated mechanisms are needed to document and manage the variability in the portfolio. One of the core mechanisms

is the variability model. It structures and describes configuration options. The frontrunner in variability modelling is the feature model (FM). An FM captures configuration options in a hierarchy of features whose selection is ruled by numerous constraints.

By its nature, the FM is a fundamental element of the configuration process. It is the base layer on which the configuration frontend relies to present options to users and determine their valid combinations. All the valid combinations are, however, rarely explicitly computed. Even in medium-size projects, millions of valid combinations can exist. Modern configurators prefer an interactive approach where user decisions are validated and propagated on-the-fly to avoid incorrect product specification. The formal semantics of FMs enables straightforward translations into established standards like propositional or predicate logics, which are supported by highly optimized solvers. These solvers are the backbone of decision validation and propagation. The process that uses an FM to pilot and validate user decisions during product configuration is called *feature-based configuration*.

In practice, applications of feature-based configuration go well beyond software tailoring. The FM is a generic constraint language designed to facilitate variability modelling regardless of the application domain. Consequently, feature-based configuration lends itself to any decision-making problem expressible in that formalism. Before delving into technical details, let us provide a first frame of reference with an example of feature-based configuration in action.

1.2 When the Rubber Hits the Road

Founded in 1898, *Renault* is a French car manufacturer serving 118 countries with personal and commercial vehicles for a total of 10^{21} variants [AD11]. A complete catalogue printed in volumes containing 5 000 000 variants in a 4cm-thick phone book format would give a pile the size of the distance between the Earth and Pluto.[1] Finding the vehicle that matches all customer requirements in such a catalogue would be quite an adventure! The duty of the configuration system is to make the selection of a product easily achievable by human beings.

For the sake of illustration, let us step in the shoes of a British shuttle company willing to buy a fleet of passenger transportation vans. To obtain a quote, the purchasing manager uses the online configurator provided by Renault, chooses the model he is interested in, and starts selecting some options. After a few decisions, he reaches the partial configuration shown in Figure 1.1.[2]

[1] The NASA estimates the maximum distance between the earth and Pluto to 7 528 000 000 km.

[2] Screenshot of `http://www.renault.co.uk/vehicleconfigurator/model/traficcpv/selectoptions.aspx` captured on July 4, 2011. For concision, only a subset of the options is shown here.

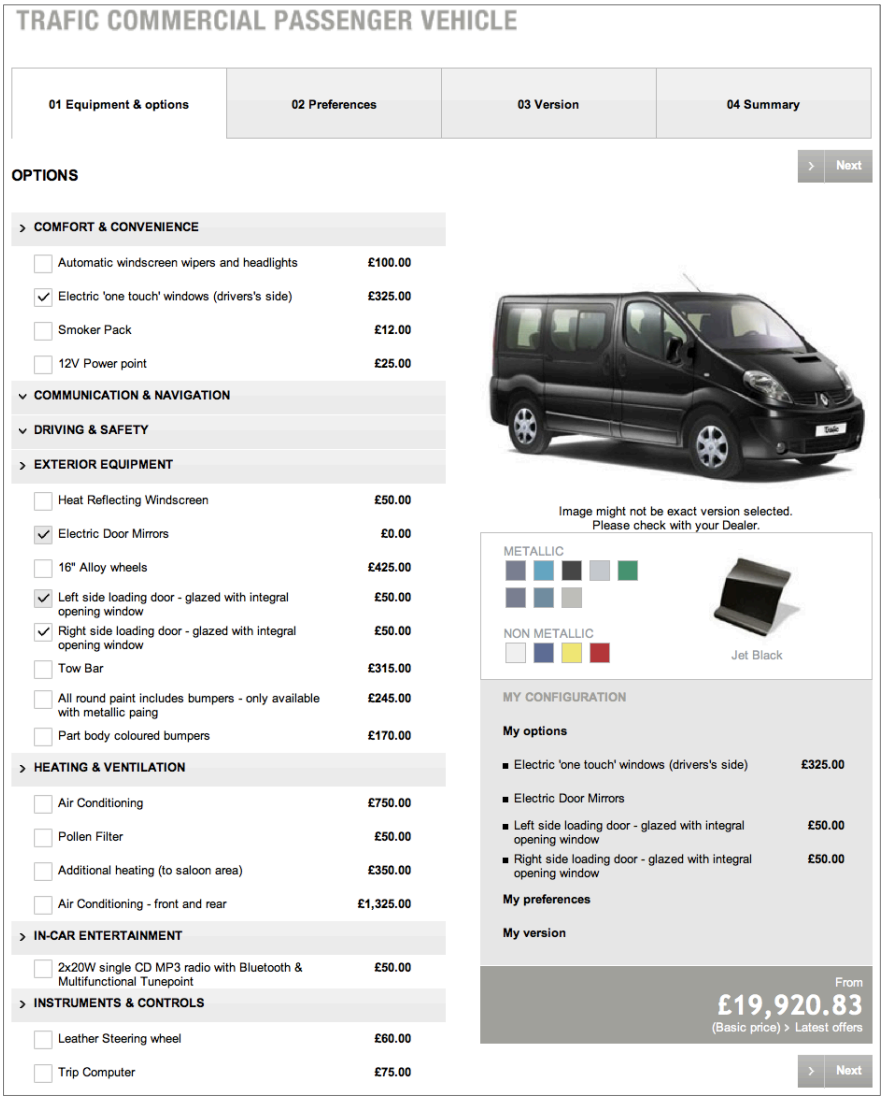


Figure 1.1 – Example of configuration menu for the Renault *Trafic*.

The policy of the shuttle company imposes dark colours for its vehicles, hence the selection of the metallic *Jet Black* colour. To prevent passengers from tampering with windows during a ride, he selects the *Electric 'one touch' windows*

(driver's side) feature. That choice results in the automatic selection of *Electric Door Mirrors*, which is a pleasant surprise for the manager. Indeed, that latter option was not available before and is automatically included. Then, he selects the *Right side loading door*, which triggers the selection of the *Left side loading door*. The purchasing manager thus makes decisions about options one at a time, rather than considering all their combinations.

This configuration panel is only the first one in a series of four. Once he is done with the selection of equipment and options, the purchasing manager can proceed to the second stage (*02 Preferences*) and so on until the final stage is reached. Besides constraints between features, constraints are also enforced between stages. For instance, to move from *03 Version* to *04 Summary*, the manager must chose one model version. At the end of the fourth stage, the configuration is sent to a car dealer that will process the demand and return a final quote.

Functionalities like cost minimisation, quote generation, order, and stock management are beyond the scope of feature-based configuration. These are domain-specific extensions wrapped around the FM that glue it together with the assets of a global system. The boundary of feature-based configuration stops at the creation of complete and correct product specifications.

This scenario is an everyday example of feature-based configuration. Yet, it already highlights how decisions progressively narrow down the decision space and can reveal hidden options, how dependencies between features restrict choices, how options are clustered into coherent subsets (subgroups of options and panels), and how configuration stages enforce the order in which decisions are made.

For the configuration frontend, Renault has chosen a web interface. Others rely for instance on tree-like representations or contextual menus. This work looks beyond the graphical layer proposed to users. The graphical syntax presented in the following examples is used for illustration purpose only and is not meant to be prescriptive. Our challenge is to define and build the underlying mechanisms that enable reasoning and assist users during configuration.

1.3 Problem Statement

The Renault example brings up two fundamental properties of feature-based configuration: *dependability* and *control*. Dependability denotes the reliability of the process. Not only must it always complete, but it must also produce a perfectly valid specification. Control indicates the ability to isolate coherent subsets of features, and to schedule their configuration. A third property, undisclosed in this example, is the *collaborative* nature of configuration. In large organisations, several actors with different responsibilities, skills, and goals repeatedly interact along the process.

Since the seminal work on feature-oriented domain analysis, FMs shifted from a static specification to a reasoning unit in interactive configuration environments. This required a whole new set of technologies on top of FMs. Decades of research have already provided efficient solvers (e.g., SAT, BDD, and CSP) that propagate decisions, ensure the global consistency of the final product, and enable automated analyses. These solvers have been successfully integrated in many commercial and open source feature-based configuration tools.

However, such tools usually do not fit well to contexts in which feature-based configuration has to be performed by multiple users or scheduled in a specific (non linear) manner. Without the appropriate support, configuration can become very cumbersome and error-prone, e.g., if a single stakeholder has to decide on behalf of all others. Furthermore, the absence of scheduling of configuration tasks makes it impossible to control the chronological dependencies between them. Finally, they barely support conflict detection and resolution.

Some might righteously argue that dedicated configurators like the one from Renault, or those used to build content management systems (e.g. Plone or Drupal), and operating systems (e.g. the Linux kernel) already partly or fully address these problems. However, these solutions are mostly *ad hoc*, come with no proof of completeness and correctness, and can hardly be reused from one domain to the other. A general and formal foundation for collaborative feature-based configuration is still missing. To tackle that problem, this thesis addresses four research questions left unanswered by related work. Several of these questions are refined in the next chapters as we dig deeper into each problem:

RQ1 *How can separation of concerns be achieved in FMs?*

RQ2 *How can the configuration process be modelled and scheduled?*

RQ3 *How can the satisfiability and completion of the configuration process be ensured?*

RQ4 *How can conflictual user decisions be handled?*

1.4 Claimed Contribution

The main contribution of this thesis is a sound foundation for collaborative, dependable, and controlled feature-based configuration. Specifically, it consists of:

C1 *A systematic investigation and understanding of the meaning of concern in FMs.* Although FM languages have been studied intensively, some questions remain open regarding their purpose. We start with a systematic

survey of various concerns FMs can address and ways in which concerns have been separated. The survey shows how the vagueness in the purpose of FM languages creates both conceptual and practical limitations.

C2 *A lightweight and flexible mechanism to leverage multidimensional separation of concerns in FMs.* From the conclusion drawn in the survey, we propose a generic technique to specify concerns in FMs, and to generate configuration views. Three alternative visualisations are proposed.

C3 *Full-fledged support for process-driven configuration.* To schedule the configuration of each view on the FM, we first formally study multi-level staged configuration, show its limitations, and use YAWL, a state-of-the-art workflow language, to define non-trivial configuration processes. We formally define a new combined formalism called feature-based configuration workflows (FCWs).

C4 *Automated analyses of feature-based configuration workflows.* The semantics of feature-based configuration workflows defines the properties of a valid execution of the workflow. However, not all executions of the workflow are semantically valid. For instance, an execution could terminate with an incomplete configuration or stall because the current decisions do not satisfy some condition. To prevent such problems, undesired behaviours should be diagnosed early and prohibited at execution time. We address this problem by defining and implementing satisfiability and proper completion analyses.

C5 *Detection and resolution techniques for conflictual user decisions.* Decision revision can lead to conflicts. In essence, a conflict denotes a decision that either violates a constraint or contradicts another decision. The challenge is to detect these conflicts and provide the user with alternative solutions, a.k.a *fixes*, to repair them. We propose an algorithm for conflict detection and range fix generation based on the HS-DAG algorithm used in model-based diagnosis. We also discuss how this algorithm can be used in a collaborative environment.

C6 *A complete implementation of all the definitions, properties and analyses in a toolset.* Support for feature-based configuration has been implemented by extending and integrating two third-party tools: SPLOT and YAWL. Natively, SPLOT supports FM modelling and configuration. We extend it to support view creation, configuration, and view-to-workflow mapping. Workflow design, execution, analysis and user management is provided by YAWL. Interactive services were added to YAWL so as to trigger view-based configuration in SPLOT. The cornerstone of this new configuration environment is the FCW Engine. Its role is to manage configuration sessions, convey the information between YAWL and SPLOT,

and monitor the whole configuration process. We also explain how our analyses fit in these three components.

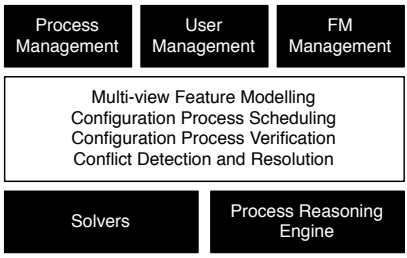


Figure 1.2 – Overview of the contribution.

As Figure 1.2 shows, our contribution is independent of the process reasoning engine, solvers, as well as process, user and FM management environments. They are viewed as black boxes on which we capitalise. Without loss of generality, our contribution applies to other variability modelling languages (e.g., CDL and OVM) that hierarchically structure the available options and guard their selection with arbitrary constraints. Our toolset extensively reuses existing third-party software to showcase the applicability of our work in practice.

1.5 Reader's Guide

Chapters are organised in layers, as shown in Figure 1.3. The research question addressed in a chapter is indicated in the left-hand side of the box.

The darker layers at the bottom are the foundations on which our contributions relies. **Chapter 2** and **Chapter 3** both revisit work related to feature modelling. **Chapter 2** recalls the essence of software product line engineering, briefly reviews existing variability modelling techniques, recalls the formal semantics of FMs, and elaborates more on feature-based configuration. **Chapter 3** is presented as an essay that discusses feature modelling from the perspective of the two leading fields in configuration: artificial intelligence in manufacturing and software configuration management.

In **Chapter 4**, we present the results of a systematic literature survey on separation of concerns in feature modelling (**C1**). We have identified seven areas of research in which concerns are discussed. Our findings show that the inherent vagueness in the feature abstraction and in the purpose of FMs makes realistic FMs hard to comprehend and analyse. Also, separation and composition techniques for concerns in FMs have been found to be rudimentary.

Based on those conclusions, **Chapter 5** reexamines the generic concept of view, provides a formal definition, investigates its properties, and proposes

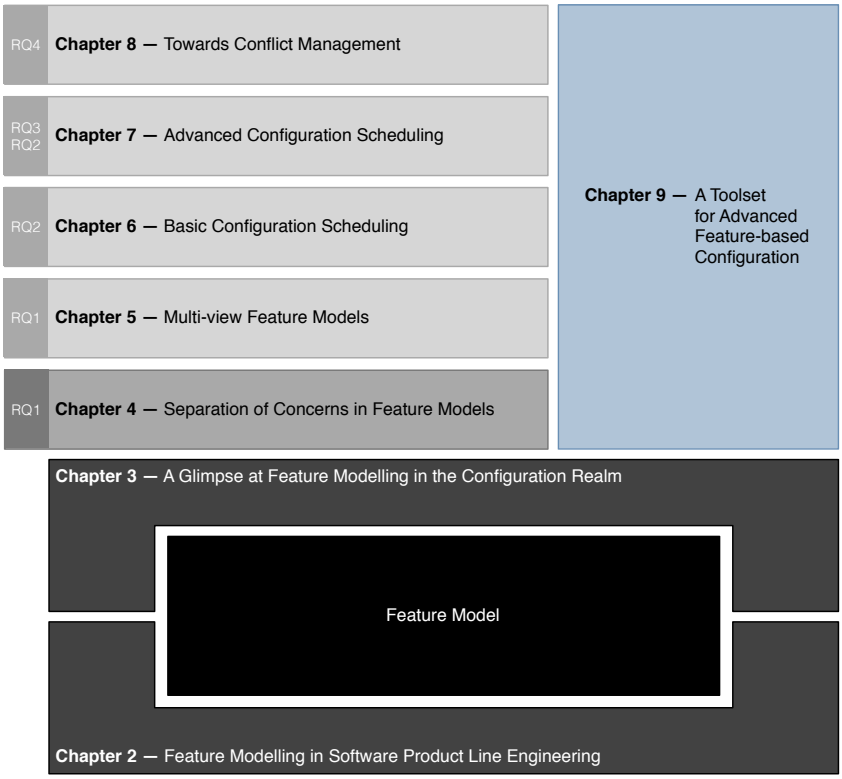


Figure 1.3 – Thesis map.

three alternative visualisations (**C2**). These techniques are motivated and illustrated through excerpts from a web-based meeting management application.

To organise these views in a coherent process, **Chapter 6** looks into multi-level staged configuration, extends the formal semantics of Chapter 2 to account for the dynamics of the configuration process, and studies its properties (**C3**).

Multi-level staged configuration has several theoretical and practical limitations. **Chapter 7** overcomes most of these limitations by coupling the FM and its views with a workflow that allows to model non-linear processes and pilot the configuration process (**C3**). That work is motivated and illustrated through a configuration scenario taken from the aerospace industry. Additionally, to avoid inconsistent and unsatisfiable models, design and execution time analyses are formally defined and implemented (**C4**). Experiments evaluate the performance of our algorithms on real-world workflows and FMs, and study the

impact that specific constructs (i.e., loops and *or*-joins) have on analyses.

These chapters either assume that all decisions are immutable or that concurrent user decisions are interleaved, thereby preventing conflicts. **Chapter 8** relaxes these assumptions and proposes algorithms that detect inconsistencies and suggest fixes (**C5**). These algorithms are evaluated with a configurator used to configure operating systems.

Finally, **Chapter 9** presents the architecture of our toolset, and details how the contribution of each chapter is integrated in YAWL, SPLOT, and the FCW Engine (**C6**).

1.6 Bibliographical Notes

The research presented in this thesis reuses and extends publications of the author. We list below the most relevant peer-reviewed papers:

Journal

1. A. Hubaux, P. Heymans, P.-Y. Schobbens, D. Deridder, and E. Abbasi. Supporting multiple perspectives in feature-based configuration. *Software and Systems Modeling (SoSyM)*, 2011. Springer Berlin / Heidelberg. (**Chapter 5**)

Conference

2. A. Hubaux, P. Heymans, and D. Benavides. Variability modelling challenges from the trenches of an open source product line reengineering project. In *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pages 55–64, Limerick, Ireland, 2008. IEEE Computer Society. (**Chapter 5**)
3. T. T. Tun, Q. Boucher, A. Classen, A. Hubaux, and P. Heymans. Relating requirements and feature configurations: A systematic approach. In *Proceedings of the 13th International Software Product Lines Conference (SPLC'09)*, pages 201–210, San Francisco, CA, USA, 2009. ACM Press. (**Chapter 4, 5**)
4. A. Hubaux, A. Classen, and P. Heymans. Formal modelling of feature configuration workflow. In *Proceedings of the 13th International Software Product Lines Conference (SPLC'09)*, pages 221–230, San Francisco, CA, USA, 2009. ACM Press. (**Chapter 7**)
5. A. Hubaux, P. Heymans, P.-Y. Schobbens, and D. Deridder. Towards multi-view feature-based configuration. In *Proceedings of the*

16th International Working Conference on Requirements Engineering:Foundation for Software Quality (REFSQ'10), pages 106–112, Essen, Germany, 2010. Springer-Verlag. (**Chapter 5**)

6. A. Hubaux, Q. Boucher, H. Hartmann, R. Michel, and P. Heymans. Evaluating a textual feature modelling language: Four industrial case studies. In *Proceedings of the 3rd International Conference on Software Language Engineering (SLE'10)*, volume 6563, pages 337–356, Eindhoven, The Netherlands, 2010. Springer Berlin / Heidelberg. (**Chapter 2**)
7. E. Abbasi, A. Hubaux, and P. Heymans. A toolset for feature-based configuration workflows. In *Proceedings of the 15th International Software Product Lines Conference (SPLC'11)*, pages 65–69, Munich, Germany, 2011. IEEE Computer Society. (**Chapter 9**)

Workshop

8. A. Classen, A. Hubaux, and P. Heymans. A formal semantics for multi-level staged configuration. In *Proceedings of the 3rd International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'09)*, pages 51–60, Sevilla, Spain, 2009. (**Chapter 6**)
9. H. Unphon, Y. Dittrich, and A. Hubaux. Taking care of cooperation when evolving socially embedded systems: The plonemeeting case. In *Proceedings of the Workshop on Cooperative and Human Aspects of Software Engineering (CHASE'09), collocated with ICSE'09*, pages 96–103, Vancouver, BC, Canada, 2009. IEEE Computer Society. (**Chapter 5**)
10. A. Hubaux, Y. Xiong, and K. Czarnecki. A survey of configuration challenges in Linux and eCos. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'12)*, Leipzig, Germany, 2012. ACM Press. (**Chapter 8**)

Doctoral Symposium

11. A. Hubaux and P. Heymans. On the evaluation and improvement of feature-based configuration techniques in software product lines. In *Proceeding of the 31st International Conference on Software Engineering (ICSE'09), Companion Volume*, May 2009. Doctoral Symposium. (**Chapter 1**)

Under review

12. A. Hubaux, T. T. Tun, and P. Heymans. Separation of concerns in feature diagram languages: A systematic survey. *ACM Computing Surveys*, 2011. (**Chapter 4**)

13. Y. Xiong, A. Hubaux, S. She, and K. Czarnecki. Generating range fixes for software configuration. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, Zurich, Switzerland, 2012. IEEE Computer Society. (**Chapter 8**)

Chapter

2

Feature Modelling in Software Product Line Engineering

2.1 Motivation and Basic Principles of SPLE

Managing a set of programs is a known problem since the early ages of software engineering. Operating system developers rapidly learned that to manage different versions of the same operating system, it is more profitable to study the properties shared by several versions before their individual details [Par76]. The A-7E Corsair II attack aircraft used by the U.S. Navy from the 60s until the late 80s was one of the first aircraft equipped with an onboard computer system. The onboard software was expected to satisfy real-time performance and modifiability constraints related to weaponry, platform, and symbology on the display. At that time, a highly dependable and modular software architecture proved to be the only way to go [BCK03]. In the financial realm, software is extensively used to track and analyse the stock market. When Market Maker committed in the late 90's to launch a web-based version of its product, it was confronted with a maze of heterogeneous databases, computing platforms, content-providing applications, and requirements. Besides these technical challenges, a very short time-to-market was mandatory to keep ahead of the competition. Their solution, called *MERGER*, was a global system satisfying all their customer requirements from which individually tailored products could be derived [CN01].

To a greater extent, the time when the industry could rely on a limited set of products with few options has passed. The diverse requirements of customers, and the necessity to differentiate themselves drove software companies to change their stance and develop versatile *product families*.

A first pragmatic intuition of product family was given by Parnas back in

1976. According to him, a set of programs should be considered as a program family "*whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members*" [Par76]. The original scope of product family was later extended to englobe marketing and managerial concerns, and cast what is today called *software product line engineering* (SPLE). A software product line (SPL) is commonly defined as *"a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission developed from a common set of core assets in a prescribed way"* [CN01]. In essence, SPLE provides support for variations in demand, operating environment, and the natural evolution of software by institutionalising *reuse* throughout software development [PBvdL05].

Systematic reuse requires high upfront investment and forward-thinking to achieve economies of scale throughout the development life-cycle. The observed benefits are usually reduced production and maintenance costs, shorter time-to-market, and more flexible response to market changes [CN01]. Several success stories of SPLE in various industry sectors have already been reported in the literature [BHJ+03, vdLSR07, Con11].

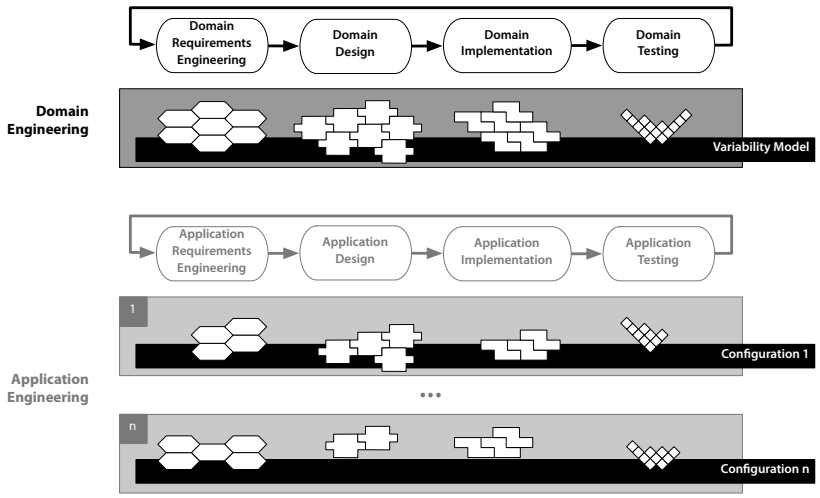


Figure 2.1 – SPLE life-cycle (adapted from [PBvdL05]).

One of the main ideas behind SPLE is to dedicate a specific process, named *domain engineering*, to the development of reusable artefacts, a.k.a *core assets*. These core assets are defined at every stage of the life-cycle, as illustrated in the upper part of Figure 2.1. The set of products that can be derived from these artefacts defines the *portfolio* of the SPL. Planning, organizing and building

a portfolio demands a rigorous documentation of the variability in the SPL. Variability is captured in a *variability model*. There exists a wide variety of variability modelling languages, as outlined in Section 2.2. For now, it suffices to know that a variability language allows to document and structure *variation points* in the SPL. Variation points are *delayed design decisions* that make core assets flexible and customizable [VGBS01].

Core assets are then reused extensively during product development, a.k.a. *application engineering* [PBvdL05]. Application engineering exploits the shared characteristics of the portfolio by *configuring* the core assets. The *configuration process* is the decision activity during which users decide which variants in the variability model have to be selected to match the requirements of the desired product. The resulting specification is called a *configuration*. The lower part of Figure 2.1 sketches how distinct configurations characterise different combinations of assets.

The different steps of the development life-cycle also suggest the notion of time (e.g., design and implementation) at which a variant is bound, i.e., selected. *Binding time* has always been a major concern in decision making. Several authors have proposed classifications [KCH+90, VGBS01, DFdJV03, vdH04, SvGB05] reflecting their experience in different industry sectors. In practice, the core assets and variability realisation mechanisms [SvGB05] supported by the SPL prescribe which binding times are used during application engineering. Binding times then define external constraints on the variability model that determine *when* decisions can be made [DFdJV03, SvGB05].

Time, however, is subject to different interpretations. The above intuition assumes that artefacts do not change. Variability is progressively resolved until the product is *complete*, i.e., there is no variability left. The variability of a given set of artefacts at a given moment is often called *variability in space* [PBvdL05]. Orthogonally, *variability in time* [PBvdL05] determines the changes to the artefacts induced by the evolution of the SPL. Section 3.3 will review some work on versioning related to variability in time. This work concentrates on variability in space. The next section revisits the most prominent variability modelling languages used in SPLE.

2.2 Variability Modelling

For SPLE to pay off, proper variability management is mandatory. Variability can be either *integrated* into artefacts such as UML models [HP03, ZHJ04, GS08] or described *independently* of, a.k.a. *orthogonally* to, artefacts.

The frontrunner in that latter category is the *feature model* (FM) [KCH+90]. First introduced by Kang *et al.* in their work on feature-oriented domain analysis (FODA) [KCH+90], an FM is a graphical representation of similarities and differences in a product portfolio. Similarities and differences in an SPL are

expressed in terms of features. A feature received various meanings over the years [CHS08]. In what follows, we will stick to the intuitive notion of feature as *"an increment in product functionality"* [BBRC06]. An FM defines a hierarchy of features and constraints on their selection. The valid combinations of features define the *products*, also called *configurations*, provided by the SPL. In other words, each product is a valid instance of the FM.

The independence of FMs from base models and code does not make it an outcast, though. Several authors have defined mappings from FMs to different kinds of base models. For instance, Czarnecki *et al.* [CP06] separate FMs from the base UML model. Classen *et al.* [CHS+10] formally relate an FM to a behavioural model and reason about them both. Heidenreich *et al.* [HSS+10] map FMs to other models such as use cases, class diagrams or statecharts. Closer to implementation, *feature-oriented programming* (FOP) has been promoted as an incremental development technique for complex programs [BSR04]. It is supported by several tools like [BSR04, KAK08, BCH+10] that provide different code generation mechanisms at different levels of granularity.

The orthogonal variability model (OVM) [PBvdL05] is another language that models variation points individually, and links them to base models. Asikainen *et al.* [ASM04] present an hybrid approach that extends the component and architecture description language Koala with variability mechanisms. The resulting combined modelling language, called Koalish, is a standalone language for describing configurable software product families.

Originally introduced in the *Synthesis method* [Cor93], *decision models* document the variability of a product family and guide application engineering. While FODA initially aimed at representing the commonality and variability of a domain, decision modelling approaches focused on product derivation. Schmid *et al.* propose a comparison of five decision modelling approaches that underlines their differences [SRG11]. In [CGR+12], Czarnecki *et al.* build upon that comparison to conduct a systematic comparison of FMs and decision models along ten dimensions. Their study concludes that both FMs and decision models have now reached comparable levels of expressiveness, and share similar use cases.

The reader is referred to Chen *et al.* [CABA09, CAB11] for a detailed and systematic review on existing variability modelling approaches and evidence of their application. The following sections dwell upon the concrete syntax of FMs, their semantics, their analysis, and introduces feature-based configuration.

2.3 Feature Modelling

The particular view of software systems offered by FMs is a powerful conceptual tool to characterise many of the products aimed at the mass market.

The apparent simplicity of FMs in describing feature relationships, and their suitability as a communication device for stakeholders, are often cited as an important reason for their popularity [KKL+98]. As a result, there has been a tremendous academic and practical interests in designing precise and expressive FM languages, automated reasoning tools to support the languages, and their application to real-life problems.

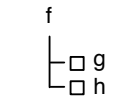
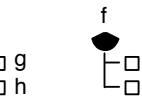
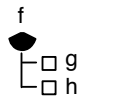
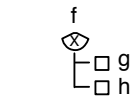
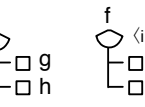
Since this original proposal, several extensions have been suggested by various authors, most of which have been surveyed in [SHTB06]. In this section, we dwell upon one particular graphical syntax, and briefly examine some textual feature modelling languages overlooked in [SHTB06]. Then, we recall the essence of the formal semantics on which this work relies.

2.3.1 Concrete syntax

Graphical representation

Most common FM languages are graphical notations based on FODA [KCH+90]. Traditionally, FMs are represented as trees whose nodes denote features and whose edges represent top-down hierarchical decomposition of features. The original graphical rendering of FMs is a tree-shaped graph which we call the *classical* concrete syntax (as shown in Figure 2.2(a)). However, here, we use a *file explorer*-like syntax because of its scalability (width grows very slowly with the number of features and complexity can be managed through "collapse and expand"). As it can be seen in Figure 2.2(b)), the file explorer syntax is much more compact than the classical syntax. Table 2.1 summarises the decomposition operators we use.

Table 2.1 – FM decomposition operators.

Concrete syntax					
Decomposition operator	and: ∧	or: ∨	xor: ⊕	generalized cardinality	optional
Cardinality	⟨n..n⟩	⟨1..n⟩	⟨1..1⟩	⟨i..j⟩	⟨0..1⟩

Let us examine these constructs before illustrating them with an example. The *and*-decomposition does not require any distinctive graphical symbol. In an *and*-decomposition, all the children that are not optional must be included in products when the parent is. The optional children must be present only if selected. Optional children are adorned with an hollow circle. It means that the selection of the parent does not imply the selection of the child. The *or*-decomposition is represented by a filled crescent spanning the feature relations.

In an *or*-decomposition, at least one child feature must be included in products where the parent is. The *xor*-decomposition is represented by an hollow crescent with an X spanning the feature relations. An *xor*-decomposition is a more constrained form of *or*-decomposition where one and only child feature can be selected. The cardinality-based decomposition is represented with an hollow crescent spanning the feature relations such that at least i and at most j child features must be present in products where the parent is.

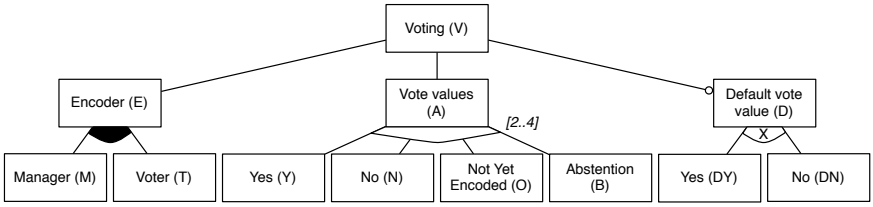
The example FM Figure 2.2 is inspired from the eVoting component of the example presented in Section 5.2. The *and*-decomposition of the root feature (*Voting*) implies that all its child features (*Encoder*, *Vote values*, *Default vote values*) have to be selected in a valid product, except *Default vote values* that is optional. The *or*-decomposition of the *Encoder* feature means that at least one of its child features has to be selected. Cardinality-based decompositions are often used, such as for *Vote values* in our example. In this case, the decomposition type implies that at least two, and at most four sub-features of *Vote values* have to be selected in a valid product. The *xor*-decomposition of *Default vote value* means that only one of its sub-features can be selected. Finally, crosscutting constraints further condition the selection of features. In the example, the selection of *Yes* under the *Default vote value* feature requires the selection of *Yes* in the vote values. The same holds for *No*.

However, working with large-scale FMs can become challenging with such notations. Given that an FM is a tree on a two dimensional surface, there will inevitably be large physical distances between features, which makes it hard to navigate, search and interpret them. Several tools have been developed to help modellers [AC04, Beu08, KTS+09, Kru07]. Most of them use directory tree-like representations of FMs to reduce physical distances between some features, and provide collapse/expand functionalities. More advanced user interfaces and visualisation techniques have also been proposed to attenuate the aforementioned deficiencies (e.g. [NOST07, CHBT10]).

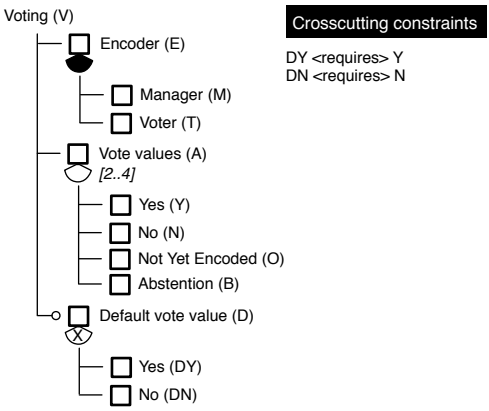
Textual representation

Various textual FM languages have been proposed as an alternative to graphical ones for a number of reasons. Their main advantage over graphical notations is that they do not require dedicated modelling tools; well-established tools are already available for text-based editing, transformation, versioning, etc. Furthermore, textual information and textual models can be easily exchanged, for instance by email.

To the best of our knowledge, FDL [vDK02] was the first textual FM language. It supports basic *requires* and *excludes* constraints and is arguably user friendly, but it does not include attributes, cardinality-based decompositions and other advanced constructs. It is also the first textual language with a formal semantics.



(a) Classical concrete syntax



(b) File explorer concrete syntax

Figure 2.2 – Sample FM of the PloneMeeting eVoting component.

The AHEAD [Bat05] and FeatureIDE [KTS+09] tools use the GUIDSL syntax [Bat05], where FMs are represented through grammars. The syntax is aimed at the engineer and is thus easy to write, read and understand. However, it does not support decomposition cardinalities, attributes, hierarchical decomposition of FMs and has no formal semantics.

The SXFM file format is used by SPLOT [MBC09]. While XML is used for metadata, FMs are entirely text-based. Its advantage over GUIDSL is that it makes the tree structure of the FM explicit through indentation. However, except for the hierarchy, it has the same deficiencies as GUIDSL.

The VSL file format of the CVM framework [Rei09, AJL+10] supports many constructs. Attributes, however, cannot be used in constraints. The Feature Modelling Plugin [AC04] as well as the FAMA framework [BSTRC07] use XML-based file formats to encode FMs. Tags make them hard to read and write by engineers. Furthermore, none of them proposes a formal semantics.

Clafer (class feature reference) is a meta-modelling language for meta-models, feature models, and combinations thereof [BCW10]. Clafer aims at

integrating feature modelling into class modelling, while providing a general framework to reason about both types of models. In particular, (partial) product configurations can be expressed in Clafer through specialisation and extension layers. Clafer also has a *"translational semantics"* to Alloy [BCW10].

TVL (Textual Variability Language) was proposed as a textual alternative targeted to software architects and engineers [BCFH10, CBH11]. The language supports four types of attributes, guards, aggregation functions, expressions, and complex constraints. It has been fully formalised, and a reference implementation is available online [CBH11]. An empirical evaluation on four industrial case studies can be found in [HBH+10].

2.3.2 Formal semantics ($[\![.]\!]_{FM}$)

Schobbens *et al.* [SHTB07] gave a generic formal semantics to a wide range of FM dialects. The full details of the formalisation cannot be reproduced here, but we need to recall the essentials. The formalisation was performed following the guidelines of Harel and Rumpe [HR00], according to whom each modelling language L must possess an unambiguous mathematical definition of three distinct elements: the *syntactic domain* $\mathcal{L}_L$, the *semantic domain* $\mathcal{S}_L$ and the *semantic function* $\mathcal{M}_L : \mathcal{L}_L \to \mathcal{S}_L$, also traditionally written $[\![\cdot]\!]_L$.

Our FM language will be simply called FM, and its syntactic domain is defined as follows.

Definition 2.1 **Syntactic domain $\mathcal{L}_{FM}$**

$d \in \mathcal{L}_{FM}$ is a 6-tuple $(N, P, r, \lambda, DE, \Phi)$ such that:

- *N is the (non empty) set of features (nodes).*
- *$P \subseteq N$ is the set of primitive features.*
- *$r \in N$ is the root.*
- *$DE \subseteq N \times N$ is the decomposition relation between features which forms a tree. For convenience, we will use children(n) to denote $\{n' \mid (n, n') \in DE\}$, the set of all direct sub-features of n.*
- *$\lambda : N \to \mathbb{N} \times \mathbb{N}$ indicates the decomposition type of a feature, represented as a cardinality $\langle i..j \rangle$ where i indicates the minimum number of children required in a product and j the maximum. For convenience, special cardinalities are indicated by the Boolean operator they represent, as shown in Table 2.1.*
- *Φ is a formula that captures crosscutting constraints (e.g. <requires> and <includes>). Without loss of generality, we consider Φ to be a conjunction of Boolean formulae on features, i.e. $\Phi \in \mathbb{B}(N)$, a language that we know is expressively complete wrt. $\mathcal{S}_{FM}$ [SHTB07].*

■

Furthermore, each $d \in \mathcal{L}_{FM}$ must satisfy the following well-formedness rules:

- r is the root: $\forall n \in N(\nexists n' \in N \bullet (n', n) \in DE) \Leftrightarrow n = r$;
- DE is acyclic: $\nexists n_1, .., n_k \in N \bullet (n_1, *), (*, n_k), (n_k, n_1) \in DE$;
- Terminal nodes are $\langle 0..0 \rangle$-decomposed.

Definition 2.1 is actually a formal definition of the graphical syntax of an FM such as the one shown in Figure 2.2; for convenience, each feature is given a name and a one-letter acronym. In abstract form, the FM of Figure 2.2 translates to:

$\mathbf{N} = \{V, E, M, T, A, Y, N, O, B, \dot{D}, D, DY, DN\}$;
$\mathbf{P} = \{V, E, M, T, A, Y, N, O, B, D, DY, DN\}$; $\mathbf{r} = V$;
$\mathbf{DE} = \{(V, E), (E, M), (E, T), ...\}$; $\lambda(V) = \langle 3..3 \rangle$; $\lambda(E) = \langle 1..2 \rangle$; $\lambda(M) = \langle 0..0 \rangle$;
$\lambda(T) = \langle 0..0 \rangle$; $\lambda(A) = \langle 2..4 \rangle$; $\lambda(\dot{D}) = \langle 0..1 \rangle$; $\lambda(D) = \langle 1..1 \rangle$; ...
$\mathbf{\Phi} = (DY \Rightarrow Y) \wedge (DN \Rightarrow N)$

Feature $\dot{D}$ is a non-primitive feature that encode optionality (adorned with small hollow circles in the concrete syntax). Every optional feature has a $\langle 0..1 \rangle$ cardinality, i.e. $\lambda(\dot{D}) = \langle 0..1 \rangle$. This is a purely technical trick in the translation from concrete to abstract syntax. It has no incidence on the user notation.

The semantic domain formalises the real-world concepts that the language models, and that the semantic function associates to each model. FMs represent SPLs, hence the following two definitions.

Definition 2.2 **Semantic domain $\mathcal{S}_{FM}$**

$\mathcal{S}_{FM} \triangleq \mathcal{P}(\mathcal{P}(P))$*, indicating that each syntactically correct diagram should be interpreted as a product line, i.e. a set of* configurations *or* products *(set of sets of primitive features).* ■

Definition 2.3 **Semantic function $[\![d]\!]_{FM}$**

Given $d \in \mathcal{L}_{FM}$, $[\![d]\!]_{FM}$ returns the valid feature combinations $FC \in \mathcal{P}(\mathcal{P}(N))$ *restricted to primitive features: $[\![d]\!]_{FM} = FC_{\downarrow P}$, where the valid feature combinations FC of d are those $c \in \mathcal{P}(N)$ that:*

- *contain the root: $r \in c$,*
- *satisfy the decomposition type: $n \in c \wedge \lambda(n) = \langle i..j \rangle \Rightarrow i \leq |children(n) \cap c| \leq j$,*
- *justify each feature: $n' \in c \wedge n' \in children(n) \Rightarrow n \in c$,*

- *satisfy the additional constraints:* $c \models \Phi$.

■

The projection operator used in Definition 2.3 will be used throughout the thesis; it is defined as follows.

Definition 2.4 **Projection** $A_{|B}$

For two given sets A and B, we note $A_{|B}$ *the projection of A on B such that:*

$$A_{|B} \triangleq \{a' | a \in A \wedge a' = a \cap B\} = \{a \cap B | a \in A\}$$

■

Considering the previous example, the semantic function maps the diagram of Figure 2.2 to all its valid feature combinations, three of which are listed below:

$$\{\{V, E, M, A, Y, N\}, \{V, E, M, A, Y, N, D, DY\}, \{V, E, M, A, Y, N, D, DN\}, \dots\}$$

As shown in [SHTB07], this language suffices to retrospectively define the semantics of most common FM languages. The concepts of *feature attribute* [KCH+90], *feature reference* [CHE04], and *feature cardinality* [CHE04], however, cannot entirely be captured by the above semantics. The extension of $\mathcal{L}_{FM}$ to support feature cardinalities is still work in progress, as reported in [MCHB11]. Attributes have already been integrated in the semantics elsewhere [CBH11]. In the remainder, d denotes an FM, and (N, P, r, λ, DE, Φ) the respective elements of its abstract syntax. Benefits, limitations and applications of the above semantics have been discussed extensively in [SHTB07].

2.4 Reasoning about Feature Models

A formal semantics is the primary measure for precision and unambiguity, and an important prerequisite for reliable tool-support. This section revisits some of the most popular analyses on FMs and reasoning techniques.

2.4.1 Analyses

The benefit of defining a semantics before building a tool is the ability to reason about the *analyses* the tool should conduct on a pure mathematical level, without having to worry about their implementation. These analyses are mathematical properties defined on the semantics that can serve as indicators, validity or satisfiability checks. Over twenty years of existence, a great deal of analyses on FMs have been proposed, the most frequent of them being:

Satisfiability. An FM d is *satisfiable* when at least one configuration, also called product, can be derived from it: $[\![d]\!]_{FM} \neq \emptyset$. An unsatisfiable FM is synonym of an over-constrained model from which no product can be derived.

Legal/valid configuration. A configuration c is *legal* or *valid* if it satisfies the FM d: $c \in [\![d]\!]_{FM}$. By extension, a *partial* configuration c' is legal if it is a subset of a legal configuration: $\exists c \in [\![d]\!]_{FM} \bullet c' \subseteq c$.

Dead feature. A feature $n \in N$ that does not appear in any configuration is called *dead*: $\nexists c \in [\![d]\!]_{FM} \bullet n \in c$.

Core features. Core features $M \subseteq N$ are features that appear in every configuration: $\forall c \in [\![d]\!]_{FM} \bullet M \subseteq c$.

Atomic set. An atomic set $M \subseteq N$ defines a group of features that always appears together in any configuration: $\forall c \in [\![d]\!]_{FM} \bullet M \cap c \neq \emptyset \Rightarrow M \subseteq c$.

Product count. How many configuration can be derived from the FM d. This is defined as $|[\![d]\!]_{FM}|$

A more comprehensive list can in found in the systematic review of Benavides *et al.* [BSRC10].

2.4.2 Automated processing

Formalising these analyses is the stepping stone to precise and reliable automated processing. The key to efficient automation is the encoding of the model and property to verify in the appropriate solver. Benavides *et al.* [BSRC10] identify four families of reasoning approaches characterised by the paradigm or method they use:

Propositional logic-based. Propositional logic is the branch of logic that studies propositions defined over a set of Boolean variables and the logical operators $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$. The abstract syntax of FMs defined in Section 2.3.2 can be readily encoded in propositional logic [Man02, Bat05, SHTB07, Tri08], as shown in Table 2.2 where

$$card_{i,j}(n) \triangleq \bigvee_{M \subset children(n) \ | \ i \leq |M| \leq j} ((\bigwedge_{m \in M} m) \wedge (\bigwedge_{m \in children(n) \setminus M} \neg m))$$

A naive encoding is used here for simplicity. Advanced discussions on optimal encodings can be found in [Sin05, MHP+07].

Two types of solvers are commonly used to efficiently handle FMs encoded as a propositional formulae. SAT solvers (e.g. SAT4J [LB11] or

Table 2.2 – Mapping from $\mathcal{L}_{FM}$ to propositional logic.

Abstract syntax	such that	Propositional logic encoding
N		Set of variables
$(n, m) \in DE$		$n \Leftarrow m$
$\lambda(n) = \langle 0..j \rangle$	$j = \lvert children(n) \rvert$	No constraint
$\lambda(n) = \langle 0..j \rangle$	$j < \lvert children(n) \rvert$	$n \Rightarrow \neg card_{j+1,\lvert children(n) \rvert}(n)$
$\lambda(n) = \langle 1..j \rangle$	$j = \lvert children(n) \rvert$	$n \Rightarrow \bigvee_{m \in children(n)} m$
$\lambda(n) = \langle j..j \rangle$	$j = \lvert children(n) \rvert$	$n \Rightarrow \bigwedge_{m \in children(n)} m$
$\lambda(n) = \langle i..j \rangle$	$1 \leq i \leq j < \lvert children(n) \rvert$	$n \Rightarrow card_{i,j}(n)$
Φ		Φ

MiniSAT [ES11]) take as input a boolean formula, and try to find an assignment of the variables such that the formula is satisfiable. SAT solvers usually require the boolean formula to be converted in a conjuctive normal form (CNF). A CNF is a conjunction of clauses where each clause is a disjunction of variables such that a variable and its complement cannot appear in the same clause. Standard conversion algorithms can be used to transform any arbitrary boolean formula into a CNF. Although satisfiability is an NP-complete problem [Coo71], SAT solvers prove to be very efficient to reason about FMs containing up to several thousand features [MWC09].

Binary decision diagram (BDD) solvers (e.g. JavaBDD [Wha11] or CrocoPat [Bey11]) encode a propositional formula as a directed acyclic graph (DAG) with two terminal nodes respectively representing `true` and `false`. Each decision node is labelled with a boolean variable, and has exactly two output edges respectively capturing the assignment of `true` or `false` to the variable.

SAT and BDD solvers differ in that SAT solvers suffer from a high complexity in time whereas BDD solver suffer from a high complexity in space. In practice, both solvers complement each other and should be picked carefully based on the type of analysis to conduct [Men09].

Constraint programming-based. A *constraint satisfaction problem* (CSP) is defined as *"a finite set of variables, each of which is associated with a finite domain, and a set of constraints that restricts the values the variables can simultaneously take"* [Tsa93]. The role of a CSP solver (e.g. JaCoP [Jac11] or Choco [Cho11]) is to find an assignment for each variable such that all the constraints are satisfied.

The mapping from an FM to a particular CSP solver is less straightforward than with propositional logic because each solver has its own encoding scheme. Details about alternative encodings can be found

in [BSRC10]. Overall, CSP solvers are less efficient to process FMs exclusively defined over Boolean domains. However, their ability to reason about arbitrary domains makes them very good candidates for attributes of numeric or textual types. They can also maximise objective functions, which is sometimes needed to optimise the cost of a set of features (e.g. total CPU consumption or memory footprint [TBC+09]).

Description logic-based. *Description logic* (DL) defines a family of formal languages meant to conceptualise, and reason about knowledge [BCM+03]. Essentially DL allow to model concepts, roles (properties on concepts and relationships among them), and individuals (instances). For instance, one of the most popular DL languages, *viz.* OWL [OWL11], has been used in combination with RACER [Moe11] to analyse FMs, and provide explanations of the results.

Other. Other approaches rely on original algorithms tailored for FMs. These algorithms are usually bound to a particular dialect of FMs and focus on very specific analyses [BSRC10]. *Satisfiability modulo theory* (SMT) [NOT06] solvers are currently being evaluated [PNX+11] as an alternative to propositional logic and CSP solvers. Solving an SMT problem means deciding whether a first-order logic formula containing predicates (such as linear equalities or inequalities) defined over non-binary variables is satisfiable.

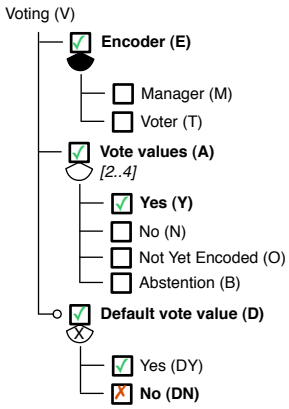
According to [BSRC10], propositional logic-based approaches (e.g., [Jan10]) are most commonly used in FM analyses followed very closely by *ad hoc* solutions (e.g., [vDK02, Men09, PNX+11]). CSP-based solutions (e.g., [BTA05]) come in third position while DL-based solutions (e.g., [WLS+07]) represent a marginal fraction of the solvers used in the surveyed studies. The growing attention for *ad hoc* solutions and the study of SMT solvers underline an interesting trend towards the development of efficient FM-specific reasoning engines.

Solvers not only help improve the quality and correction of FMs. They also provide the backbone that maintains the consistency of the configuration throughout the application engineering process.

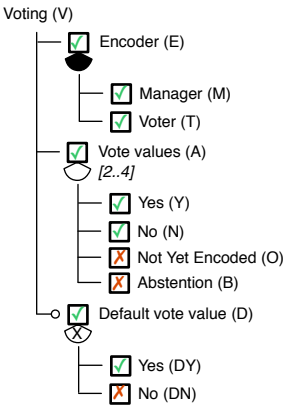
2.5 Feature-based Configuration

Product configuration is a labour-intensive and time-consuming activity that usually takes up to several months and involves numerous stakeholders with heterogeneous concerns [DS05, MCMdO08, HHSD10]. *Feature-based configuration* (FBC) is the interactive process during which stakeholders decide which features are included in a product.

The fundamental challenge faced by FBC systems is to ensure the correctness of the configuration *along* the whole process. In an interactive setting, it means (1) automatically *propagating* decisions, and (2) *explaining* the results of the propagation. *Decision propagation* is the procedure that automatically sets the values of variables that depend on the decision. In other words, feature required (respectively excluded) by the decision are automatically selected (respectively deselected). As a result, no decision made during the configuration process can ever break the satisfiability of the product. An *explanation* is the feedback delivered to stakeholders that details how a decision was made, i.e. manually or automatically. In the automatic case, the explanation should contain the manual decision that triggered the propagation, and the constraints that entailed the (de)selection. Note that several explanations can match a single decision.



(a) Manual decision (DY)
Automatic decisions (E,D,Y,A,$\neg DN$)



(b) Manual decisions (M,T,N,$\neg O$,$\neg B$)

Figure 2.3 – Examples of manual and automatically propagated decisions.

In the eVoting example, the selection of DY entails the automatic selection of D, Y and A and the deselection of DN, as shown in Figure 2.3(a). The selection of E is not a result of the selection of DY but of the structural constraint binding V and E ($V \Leftrightarrow E$). In Figure 2.3(b), the stakeholder proceeds with the configuration. All the decisions here are manual, i.e, no decision is propagated. The outcome is the complete set of features that specify the product: $\{V, E, M, T, A, Y, N, D, DY\}$. In the example, a first explanation for the selection of A is that it was selected because the manual selection of DY, which implied the automatic selection of Y ($DY \Rightarrow Y$), which in turn implied the selection of A (parent of Y). A second explanation is that A is automatically selected by construction ($V \Leftrightarrow A$).

In practice, performance, scalability, and integration requirements also prevail. FMs can contain thousands of features whose legal combinations are determined by quantities of complex constraints [Bat05, Men09, BSL+10]. That level of complexity demands extremely efficient reasoners to propagate decisions while preserving the reactivity of the configuration interface. The interface itself must also offer scalable navigation and explanation mechanisms. Furthermore, large models can also be synonym of several stakeholders who intervene at different moments in the configuration process. Finally, the uptake of an FBC system is determined by its ability to be integrated in full-fledged development environments. Concretely, it means interacting with heterogeneous modelling and programming tools to derive running products.

Over the years, tools supporting FBC capitalised on the formalisation of FMs to develop interactive configurators (e.g. [AC04, psG06, KTS+09, Men10]). These tools rely on efficient solvers (typically SAT, BDD and CSP solvers) that excel at propagating decisions throughout the FM. As we have seen in Section 2.4.2, their reasoning abilities depend on the type of solver. For instance, SPLOT [Men10] uses SAT and BDD solvers only, which restricts the reasoning domain to binary decisions but enables extremely fast reasoning about thousands of features. pure::variants [psG06], on the other hand, uses a dialect of Prolog. Prolog being a general-purpose logic programming language, it supports arbitrary domains such as integers or strings. But that gain in expressiveness comes at the price of lower overall performance. Usually, these tools also propose other analyses such as explanation, dead feature detection, core feature detection, model comparison, model metrics computation, and auto-completion.

Commercial tools (e.g.pure::variants [psG06] and GEARS [BS10b]) also offer bridge integration with off-the-shelf toolsets like IBM Rational, Rhapsody, Simulink, Serena or Visual Studio. Others like XTof and FeatureIDE [GCB+10, KTS+09, KÏ1] focus on FOP, and provide traceability between an FM and a code base. The configuration of the FM echoes with the automated pruning of the unnecessary source code. These advances place FBC at the core of software engineering environments.

2.6 Chapter Summary

Increasingly widespread in industry, SPLE is leading research on variability-intensive software development and reuse management. SPLE has incubated research on feature modelling and fostered its adoption among practitioners. This lead to its current standardisation by the OMG in the *common variability language* (CVL).[1] After a swift introduction to the motivation, goal and fundamentals of SPLE, this chapter enumerated the main variability modelling

[1] `http://www.omgwiki.org/variability/doku.php`

languages and several dialects of FMs. It then introduced the concrete syntax and recalled the semantics of FMs of Schobbens *et al.* [SHTB07] used throughout this thesis. Finally, common analyses of FMs and reasoning techniques used in FBC were described and exemplified.

Chapter

3

A Glimpse at Feature Modelling in the Configuration Realm

3.1 Overview

Since the early 90's, FBC has adopted and revisited several notions from two landmarks of computer science: *artificial intelligence* (AI) and *software configuration management* (SCM). The tight bond between configuration in AI and FBC has already been established. Günter *et al.* [GK99] recognise *concept hierarchies* (similar to FMs) as a fundamental concept in their survey of configuration methods used in knowledge-based configuration. According to Junker's classification of known configuration problems [Jun06], FBC falls in the *option selection* or *shopping list* problems. Additionally, several authors have already reviewed how AI contributes to the automation of FBC (e.g. [Men09, Jan10, BSRC10]).

Rather than repeating existing work, we take several steps back from the technical contribution of AI to outline its support for configuration in the manufacturing industry and compare it to FBC. The manufacturing domain caught our attention mainly because it has led research on configurators—more generally called *expert systems*—in AI for years. Other researchers also stressed the need to understand the relationship between hardware and software product lines (e.g. [HK07]). In fact, many reports on the applications of SPLE are very close to the manufacturing industry. A quick look at the SPL hall of fame [Con11] shows that out of the 18 success stories, only 5 are geared towards end-users. Most of them are dedicated to embedded devices like cell phones, engines or military equipment.

While configurators help users create superior products, SCM manages change along the engineering lifecycle. Since the 50's, SCM has grown into a

mature discipline, essential to most software development projects [ELvdH+05], and has been included in many IEEE, ISO, EIA, and military standards [Pre04]. The major reasons for the success of SCM are its generality and independence of application semantics, which make it " *"universally" applicable, while still providing a useful abstraction layer upon which other software engineering tools operate and integrate their results."* [ELvdH+05] To gauge the full extent of SCM, we quickly recall its basic functionalities. We then dwell upon the relationship between FBC and product configuration in SCM.

This chapter aims at raising the reader's awareness on the connection between FBC and the application of AI in manufacturing and SCM. Without seeking completeness, we highlight some key synergies and expose the bigger landscape in which FMs fit.

3.2 On the use of FBC in Manufacturing

3.2.1 Configuration in manufacturing

Like software vendors, manufacturing companies face a new generation of customers looking for *customisable products* at the same price and conditions than previously standard products. Failing to provide that extra flexibility can rapidly become a serious competitive disadvantage. Furthermore, the cheap cloning of standard products in low-cost regions has lured a significant amount of customers away from their regular dealers [Con08]. A vast and high-quality portfolio is thus a prerequisite for business sustainability.

To cut down costs and compete with low-cost manufacturers, many companies resort to *lean production* [WJR91]. Lean production is a production practice that aims at minimising the consumption of resources that do not produce any value or improve productivity. This practice, combined with the growing necessity to market customisable products, drove manufactures to adopt sophisticated configurators that reduce lead times[1], automate quote generation, increase workforce efficiency, reduce errors in bills of material, and increase control over the production [HMR08, Con08, TSMS96].

Configurators are commonly used to manage *"the sales, product design, and development of manufacturing specifications for customised products"* [HMR08]. At the core of the configurator lies the *configuration system*, which is *"an expert system that is able to combine modules which are individually described by a number of characteristics, by using rules (constraints) which describe which modules are legal to use in combination"* [HMR08].

Hvam *et al.* [HMR08] stress that the decision to create or use a configurator should not be technology-driven but rather result from the clear prevision of

[1] The lead time is the period of time between the beginning and the end of a production activity.

commercial advantages like increased customer satisfaction and market share, and reduced production costs.

Configurators not only hinge on sound technologies. Their purpose and design is determined by a delivery strategy that prescribes the degree of flexibility and their position in the manufacturing lifecyle. The four main strategies are discussed below. We then outline the configurator development process recommended by Hvam *et al.* [HMR08] that has been successfully applied to numerous large scale industrial projects (e.g. the production of complete cement factories), and then close in on the variability model they advocate for the definition of the product range.

3.2.2 Different product delivery strategies

Traditionally, companies have been fabricating, assembling and eventually storing finished products. In this *make-to-stock* delivery strategy where products are manufactured for storing, the *customer order decoupling point* (CODP), also called *order penetration point*, is positioned at the end of the production process. The CODP is the *"the point in the manufacturing value chain for a product, where the product is linked to a specific customer order"* [Olh03].

However, as the market moved from forecast-driven to customer-order-driven, the CODP progressively shifted upstream to accommodate the offer to the growing need for early customisation. Table 3.1 sketches the position of the CODP for the *make-to-stock* strategy as well as those of three other strategies, which are explained below.

As we go upstream, the first strategy we meet is *configure-to-order*, a.k.a. *assemble-to-order*. In this strategy, products are built based on a combination of standard components pieced together following some choices of the customer. A car is an example of a product assembled to meet a specific order [Con08]. One level up we find the *make-to-order* strategy, which is more flexible than *configure-to-order* as it only starts the manufacturing of already designed components once the order has been placed. It is therefore easier to build more customisable products as the fabrication of the pieces is case specific. Hardware manufacturing at Dell is an example of the application of such strategy. The final stage is the *engineer-to-order* strategy, which leaves a maximum flexibility to the customer. In that case, a substantial amount of work is needed to define accurate specifications. The production of complex plants like cement factories belong to that strategy [HMR08].

Naturally, the upper in the stream the more complex the strategy is to handle. Not only the manufacturing facilities must be flexible, but also the design and specification of the product family. In fact, to deal with that complexity, some companies combine different strategies [TSMS96]. Empirical studies have shown that, on average, 80% of the customer orders can be satisfied by configured products [TSMS96, HMR08]. The last 20% have to be handled in-

Table 3.1 – Different product delivery strategies (adapted from [Olh03]).

Product delivery strategy	**Design**	**Fabrication and procurement**	**Final assembly**	**Shipment**
Make-to-stock	- - - -	- - - -	- - - -→	CODP →
Configure-to-order	- - - -	- - - -→	CODP ⟶	⟶
Make-to-order	- - - -→	CODP ⟶	⟶	⟶
Engineer-to-order	CODP ⟶	⟶	⟶	⟶

dependently, which results in markedly higher costs and longer delivery time. Similarly, in SPLE, O'Leary *et al.* [ORRT09] report that customers are usually ready to have only 80% of their requirements fulfilled if the cost of the deviation from standard products is too prohibitive.

The nature and complexity of the products determine the type of strategy(ies) that best fits the needs of the company. It is only once agreed upon that the development of the configurator can actually start.

3.2.3 Building the configurator

According to Hvam *et al.* [HMR08], the creation, implementation and operation of a configurator is a seven-phase procedure. During the first phase, the *product specification process* is defined. The specification process is the process that analyses the needs of the customers, creates a product tailored to them and prescribes the activities related to, for instance, purchasing, delivery, servicing and recycling. Figure 3.1 illustrates how a sample specification process is integrated in the production process. The specification process also defines the configuration system that supports the activities composing it.

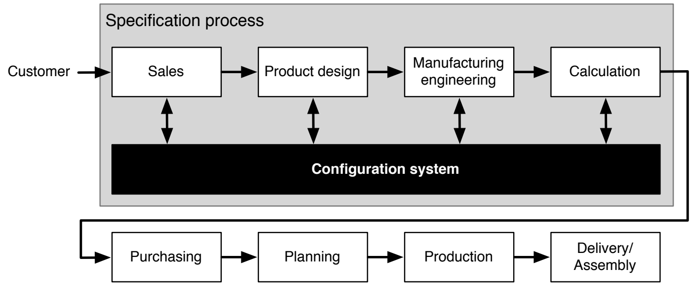


Figure 3.1 – Specification process endowed with a configuration system (adapted from [HMR08]).

The second phase is the *analysis of the product range*, which aims at providing a global view of the product range by detailing the product knowledge to be included in the configuration system. The model used to capture this knowledge is called the *product variant master* (PVM). Basically, the PVM consists of a set of hierarchically organised components and their parts. The PVM is usually drawn on a large sheet of paper or with tools like Microsoft Excel or Visio. The specifications of individual components and parts are usually documented via class, responsibility and collaboration (CRC) cards [BC89]. We stop here the description of the PVM as Section 3.2.4 is dedicated to it.

In the third phase, a first *object-oriented model*, in this case a class diagram, is directly derived from the PVM and CRC cards. Once completed, the object-oriented model contains both the product and system knowledge. This model is then adapted in the fourth phase, called *object-oriented design*, based on the selected configuration system. The configuration system contains the configuration knowledge and performs the reasoning about module combination. It can either be developed in-house or, more frequently, be supplied by a third-party. Since many of the existing systems are not fully object-oriented, the model must be adapted to work properly with the selected software. Note that the integration with more than one third-party tool might prove mandatory. For instance, the configurator might have to interact with an ERP to compute delivery times and prices, CAD tools supporting the modelling of complex physical parts like an engine or even 3D visualisation tools used by architects to present explorable representations of the future building.

The role played by FBC in SPLE reflects that of the configuration system in manufacturing. However, FBC is restricted to option selection and attribute assignment, while the configuration system covers the simplest forms of binary decisions to the assembly of complex mechanical parts. In other words, FBC can only cover a limited span of the range of problem tackled in manufacturing.

The fifth phase deals with the actual *programming* of the configurator, which is then *implemented* in the production environment during the sixth phase. The final phase, the seventh, is focused on the *maintenance and further development of the configurator*.

3.2.4 Product variant master

As said earlier, the PVM aims at capturing the various modules and parts of the product range, hence its variability. To illustrate the concepts and terminology of the PVM, we will use the example presented in Figure 3.2. The left-hand side, also known as the *part-of* structure, hierarchically defines the component of the product family. The *part-of* structure is analogous to the aggregation relation in UML. It shows notably that the *Car family* is defined by a *Windshield*, a *Gearbox*, a *Sunroof*, an *Engine*, a set of *Wheels* and the *Air conditioning*. The right-hand part, also known as the *kind-of* structure,

specifies the *variants* of each of these components. The *kind-of* structure is analogous to the specialisation relation in UML. Figure 3.2 shows only one example of *kind-of* structure, i.e. the car family can either be *Station wagon*, *Van* or *Cabriolet*. Another example, not present in the figure, is the kind of wheel available, i.e. *14" standard tire*, *15" tire* or *Super low profile tire* [HMR08].

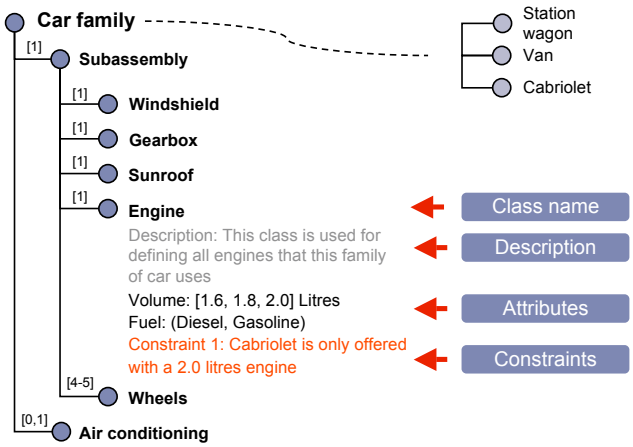


Figure 3.2 – Example of PVM (copied from [Har06]).

The concepts of the PVM are deeply inspired from the object-oriented paradigm, which greatly facilitates their conversion into class diagrams. In the PVM, every component preceded by a circle with a short horizontal line on the left is the *class name*, which must be unique. Classes can also have *cardinalities*, which specify the number of sub-parts it has. In our case, a car can have between 4 or 5 wheels if it has a spare wheel. Classes can also have a short *description* and *attributes* like the *Volume* or *Fuel* type of the *Engine* in Figure 3.2.

Four types of *constraints* can be attached to a PVM or a class. A constraint denotes how classes and attributes can be legally combined. The first type of constraint is the *verbal* constraint, which allows free-form natural language to be used. The second, the *logical* constraint, is typically expressed in a semi-formal or formal logic like propositional logic. The third type of constraints is expressed as a *calculation* over attributes of classes. The fourth type is the *combination table*, which describes constraints through relationships. For instance, it can represent the relationships between the parts available and the assemblies of which they are constituents. In that case, a cell of the table would, for example, represent the number of pistons of a given diameter needed

to build a specific engine.

A great variety of viewpoints on the product range can co-exist. According to Hvam *et al.* [HMR08], three views to describe the product range are particularly sensible in practice, where a view is an independent PVM. The *customer view* concentrates on the functions, properties and components relevant for the customer and typically contains elements of the related technical processes, interfaces with the environment and features of the product. This view explains *"what makes customers buy the product"* [HMR08]. The *engineering view* describes the relationship between the functions and components by looking at the solution side of the problem. It explains *"how the product works"* and *"which functional variants exist"* [HMR08]. The *production view* goes further down in the abstraction and focuses on the detailed components and lifecycle properties of production and assembly. It can also be an extension of the engineering view where all the low-level details are incorporated, which can result in 10 000 or more classes. It explains *"how the product is produced"* [HMR08].

Even though targeting different concerns, these views complement each other and contribute to the elaboration of concrete products. The pairwise relationships between the different views are called *causal relations*. By following the causal relations top-down, starting from the customer, to the engineering and from the engineering to the production view, one answers the question: *"How is the feature realised"* [HMR08]? Whereas bottom-up, i.e. all the way up from the production to the customer view, one answers the questions: *"Does this variant add value to the customer"* [HMR08]? The number of relations among the views gives a good overview of the complexity of the product range. It notably helps evaluate the magnitude and impact of the work needed when changes to the product range are planned.

The authors regularly punctuate their definition of the PVM by examples and reports taken from their experience. Due to space limitation, we only quote here two lessons learned that illustrate well the relevance of PVMs in practice.

> *[. . .], domain experts often find it easier to understand the product variant master* [than class diagrams and CRC cards]*, as its notation is closer to the concepts and structures the domain experts are familiar with in their daily work. [HMR08]*

> *The product variant master has created the basis for a far more professional dialogue between sales, development and production. One designer expressed it in this way: "This is the first time we have had the possibility of a meaningful dialogue with sales". [HMR08]*

Obviously, the PVM has lots in common with FM languages supporting both attributes and feature cardinalities. The FM in Figure 3.3 is a tentative

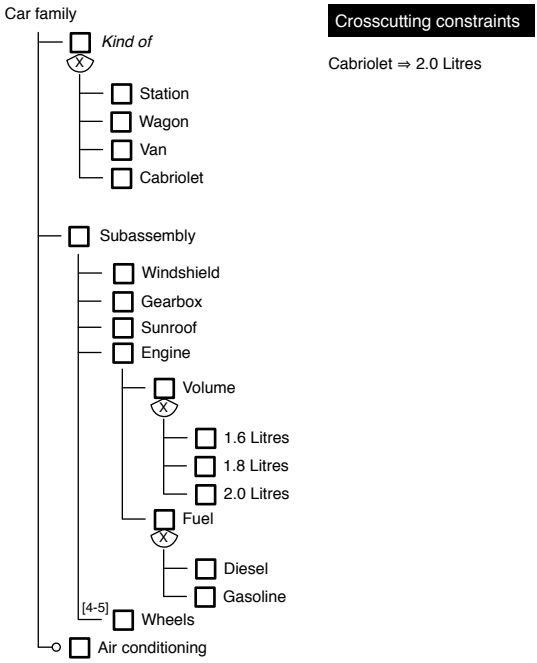


Figure 3.3 – Example of translation of the PVM in Figure 3.2 into an FM.

translation of the PVM in Figure 3.2. The textual constraint has been expressed with a boolean implication, the multiplicity is modelled with a feature cardinality, and the *kind-of* structure represented by an extra *Kind of* feature.

Yet, although the PVM was introduced almost ten years after FODA, we could not find a reference to FODA in the PVM literature, and vice versa. Albeit sharing the same motivation and issues, these two engineering fields have evolved independently. In order to identify possible bridges, we first look at FMs as a means to model the product range, and then as a basis for product configuration.

FMs as means to represent the product range. The first major difference is the semantic distinction between the *part-of* and *kind-of* structures, which are not standard FM constructs. However, comparable structures have already been proposed (e.g., [LKL02]) for FMs and could be reused here. The second major difference is the specification of constraints. Verbal constraints and combination table have not been implemented in FMs. The addition of the former would be trivial but hardly amenable to automated reasoning. The latter is a mere tabular representation of a set of constraints. Thereby, we postulate that FMs can be used as a replacement of PVMs and therefore be

a candidate to document the product range. It is interesting to note that this purpose is actually the most common one for FMs. As Chen *et al.* [CABA09] report, 23 of the 33 approaches they reviewed address the requirements phase of domain engineering, which corresponds to the role of the PVM in the lifecycle.

FMs as a basis for product configuration. Our investigations show that the PVM is not a direct interface to configure products. Instead, it is used to generate a class diagram that is subsequently refined and used to build the configuration system. The formal reasoning during the configuration is usually handed over to the expert system. In contrast, research on FBC has made FMs amenable to automated processing. While the PVM is confined to documentation purposes, FMs add a reasoning capability that can be used throughout the development and even be part of the final product if runtime configuration is supported (e.g. [DVC$^+$07]).

Unlike PVMs, FMs play a direct and explicit role in the configuration validation process. The reasons why PVMs do not serve that purpose are not clear. We can imagine that the absence of formal semantics and specific tool support are among them. Also, expert systems seem to be well accepted and traditionally used in configurators, which might rule out other forms of constraint reasoning or consistency checking. Furthermore, the constraints to deal with and calculations to perform are clearly not limited to the information present in the PVM. For instance, delivery times and production costs are not part of the PVM and yet used to produce quotations.

In that context, even if FMs were considered as an alternative representation, they would only allow to automatically generate a part of the rules existing in the expert system. The analyses commonly performed on FMs like satisfiability analysis and dead feature detection would have to be constrained by the production activity and customer requirements. For instance, a feature may not only be dead because it is not part of any valid product but also because its price or manufacturing time make all the products containing it unsaleable. The challenge is thus to integrate the FM into the knowledge base, and to automate reasoning on the whole knowledge base. Along the same line, Schmid *et al.* [SK09] observed that a stronger emphasis is put on the relationship between the artefacts (e.g., components and abstract features) in existing configuration systems than in SPLE. That trend to isolate the FM from other artefacts would probably be a barrier to its application in manufacturing.

The representation itself of the PVM could also be a reason why they are not used as a configuration interface. Although FMs and PVMs have a lot in common, the file explorer representation of FMs is regularly used as a configuration interface in research (e.g. FeatureIDE [KTS$^+$09] and SPLOT [MBC09]) and commercial (e.g. Pure::Variants [psG06]) tools, unlike the PVM. The reason why the same kind of model is considered as a proper interface for configuration in one case (FMs) but not in the other (PVMs) is still obscure. Finding an explanation is further complicated by the lack of published evidence on the

application of FMs in practice [HCMH10].

3.3 On the use of FBC in Software Configuration Management

3.3.1 Fundamentals of SCM

For the general audience, the advent of popular tools like CVS or Subversion has somehow confined SCM to software versioning. Yet, SCM has a far wider scope:

> *Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team. The goal is to maximize productivity by minimizing mistakes.* [Bab86]

SCM permeates every step of the software engineering life-cyle. It also involves many actors with distinct responsibilities, tasks, and expectations from the SCM system. For instance: the *project manager* monitors the progress of the project and makes sure that the product will be developed within the given time frame; the *configuration manager* enforces development policies (e.g. code creation, change, and testing) and collects data about the status of the project; and the *quality assurance manager* controls the quality of the product [Dar91].

The spectrum of functionalities offered by an SCM system has to cover the requirements of every actor. The most demanded functionalities are classified in Figure 3.4. A functionality wrapped in a circle indicates that it can exist by itself in the SCM system. When these self-contained functionalities are combined with the team and process functionalities (rectangular boxes), one obtains a comprehensive SCM system [Dar91]. The layering indicates that top-level boxes rely on functionalities from lower-level boxes.

A product is composed of several artefacts, also known as *configuration items*. These artefacts are identified (typically by a version number), stored and accessed with the *artefacts* functionality. How these artefacts are structured and related to each other is maintained by the *structure* functionality. The actual generation of the executable software from these structured artefacts is supported by the *construction* functionality.

The *auditing* functionality records all the changes performed, keeps track of who made those changes, why and how. Besides tracking changes, the SCM system must also gather statistics about the product and development process to enable report generation (*accounting* functionality). To preserve quality, the *controlling* functionality (1) controls change requests, changes, and problem reports, (2) propagates changes, and (3) monitors who makes what change when.

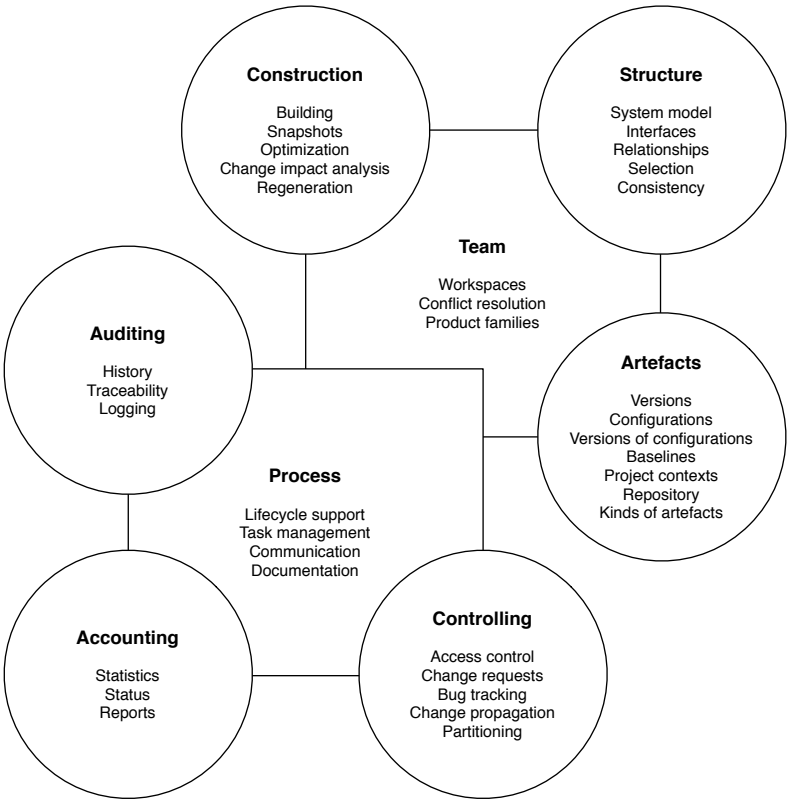


Figure 3.4 – Functionalities of an SCM system (adapted from [Dar91]).

The last two functionalities thread them all together. The *process* functionality integrates the tasks to be performed in the global engineering lifecycle, assists users in the selection of these tasks, and documents the knowledge of the process. The *team* functionality supports collaborative work through workspace definition, conflict detection and resolution, and the maintenance of product families.

Most of these functionalities will not be explained further for concision. The interested reader is referred to [Fei91, Dar91, Pre04, ELvdH+05] for an in-depth introduction to SCM, and to [Dar99] for an additional list of challenges posed by web systems. In the remainder of this section, we concentrate on the relationship between FBC and product configuration, collaborative development, and process management.

3.3.2 Product configuration

A *configuration* specifies which configuration items are needed to build a product or to populate a *workspace* [Dar91, Pre04, ELvdH+05]. A workspace *"provides users with an insulated place in which they can perform their day-to-day tasks of editing and using external tools to manipulate a set of artifacts"* [ELvdH+05].

Each configuration item is subject to change during the engineering process. They are, however, rarely versioned individually. Instead, they are commonly grouped in coherent units (e.g., packages) whose versions are organised in a *version graph*. The version graph is a DAG in which the parent-child relationship is either a *revision of* relationship (sequential development branch), a *variant of* relationship (parallel development branch), or a *merge* relationship (combination of development branches). That classic representation is called the revision/variant/merge version model. Several such graphs can co-exist (e.g., one per package) in the SCM. Some nodes of these graphs, called *baselines*, define milestones in the engineering process. A baseline is a *"specification of product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures."* [IEE]

The high number of configuration items multiplied by the ever-growing number of versions demands tools that enable the selection of versioned artefacts without having to request each of them individually. There are basically four types of selection mechanisms [ELvdH+05]. They are illustrated below with the creation of a workspace.[2]

Hierarchical workspaces. In this mechanism, each workspace represents a small increment in functionality. These workspaces are hierarchically structured to imitate the structure of the tasks to be performed. A workspace both contains local versions of some artefacts and inherits other artefacts from its ancestors.

General queries. To select artefacts and their versions imperatively, some SCM systems augment artefacts with multiple attributes that enable general queries. These queries return one or several configurations that match these criteria. The authors of Adele report from their experience that *"describing a configuration by giving the full list of its components is not appropriate [...] Experience shows that complex configurations are specified with a few lines of constraints, even if numerous modules and constraints are involved in its construction"* [BE86]. Below is an example of configuration of an interface that specifies the implementation

[2] Note that workspace creation is usually a prerequisite to product generation. The necessary configuration items are either downloaded directly in the workspace or linked before compilation.

(`m2-i1-v2.003`) that must be satisfied by several constraints: it must be in a consistent state; at least one implementation of the `m1` family must have the `unix` or `unix_v5` attribute; and the default (`seldef`) values for the type, state, and build date must be satisfied if possible—these are weak constraints that express a preference.

```
configuration m1-i-c
    selimp
       m2-i1-v2.003
    selnot
       * (state=inconsistent)
    selcond
       or (m1* (syst=unix), m1* (syst=unix_v5))
    seldef
       * (type=debug, state=official, date<85_06)
```

The PROTEUS Configuration Language is another example of formalism developed for system modelling, configuration definition, and system building. It is based on the notion of *family description* that *"encompasses all potential variability of the entity"* [GG96].

Leverage change-sets. Change-sets are deltas to a configuration item that are stored independently of the other changes. A configuration item is then constructed by composing a set of change-sets with a baseline. By extension, a workspace is composed of several change-sets and baselines.

Rule-based. This mechanism specifies a configuration with an ordered set of rules. The difference with general queries is that rules impose an order while queries are general and/or formulas. The example of rule below comes from the SHAPE SCM tool [ML88]. Intuitively, it means *"select the newest version of all components that I am working on"*, or *"select the newest published version of all other components"* if the previous rules fails.

```
exprule:
   *.c, attr (author, $(LOGNAME)), attr (state, busy);
   *.c, attrge (state, published), attrmax (version).
```

After three decades of intense development, industry has only adopted a fraction of the solutions proposed by researchers [ELvdH+05]. Only the simplest forms of change-sets and rule-based techniques have made their way in everyday practice. Users mostly prefer to build workspaces that vary little from a baseline, run frequent builds and tests, and then create a new baseline. These workspaces are easy to specify with a few rules or change-sets. Conversely, general queries can produce configurations that deviate significantly from a known baseline, which is often unmanageable for most users.

From these four mechanisms, general queries is the closest to FBC. It is also the least appreciated one in SCM, which raises the question whether FBC would be a candidate for configuration in SCM. Not only is that relevant for FBC in general, but also for change management in SPLE, a.k.a. *variability in time* [CN01, Kru02, PBvdL05, AdOM+09] (see also Section 2.1).

The fundamental problem of general queries is the lack of control over the variations between a configuration and a known baseline. By exclusively focusing on the options they are interested in, users are unable to predict the value of the ignored parameters. In FBC, a similar threat awaits textual variability modelling languages like Clafer [BCW10] (see Section 2.2) that specify configurations in the same fashion.

A simple solution to that problem is the default configuration. By selecting a default configuration, users establish a first frame of reference for the configuration process. Then, by altering the necessary options, they can reach the desired configuration while controlling the full extent of variations. That solution is actually common practice in operating system configuration [BSL+10] where the very large number (over 5000 for Linux) and complexity of the parameters make the configuration from scratch downright impossible. Modern distributions of Linux, for example, provide default configurations for all known hardware architecture (e.g. `x86`, `alpha`, and `AMD64`). Similarly, eCos [eCo11] provides default templates for every compatible board.

Furthermore, FBC pushes the boundary of general queries. As we have seen in Section 2.2, techniques like FOP can link a feature to code snippets. In that case, the granularity is no longer the file but coherent blocks inside it. To our knowledge, general queries do not support such a fine-grained level of selection. Whether this would be an actual advantage in SCM calls for empirical evaluations but it appears that FBC could be a possible candidate for product and workspace configuration in SCM.

3.3.3 Collaborative development

Early SCM systems failed mostly because they were *"helping the configuration manager, and bothering everyone else"* [Est00]. That problem has been progressively addressed through improved workspaces. We have already seen that a workspace sandboxes a set of configuration items on which the engineer can work in isolation from the shared repository. Besides editing these files safely, i.e., without risking corrupting the repository, a workspace also allows to build (link and compile) binary files.

To deal with the explosion of file copies, high-end SCM systems now propose *virtual workspaces*. The central idea behind virtual workspaces is that files are only copied on-demand, i.e., when they have to be edited. All the linking and file updates are transparently maintained by the SCM system without affecting the tasks of the user. Additional optimizations also avoid recompi-

lation of already compiled objects. For instance, pre-compiled objects can be recovered from other builds on the grid, which reduces compilation costs to a bare minimum.

At a global scale, several engineers collaborate on the same project. To optimize teamwork, SCM must support effective concurrent work, which resonates with *conflict management*. When changes in a workspace are *committed* back in the repository, conflicts are likely to occur. A big part of workspace management includes *syncing* changes, and *resolving* conflicts. Several syncing and resolution mechanisms have been developed, from the most basic like diffing to advanced semantic-based merging. As for configuration selection, commercial SCMs turned their back on the most advanced forms of merging chiefly because they introduce too much complexity. This extra complexity often produces more variants than needed as many organisations actually struggle to reduce the number of variants to *"save costs and improve quality"* [ELvdH+05].

Research on collaborative FBC and conflict detection and resolution is still in its infancy. If the need for separation of concerns is well acknowledged (see Chapter 4), the decomposition of the configuration activity into collaborative virtual workspaces is still insufficiently supported. Chapter 5 puts the emphasis on multi-perspective FBC, paving the way for insulated configuration workspaces. The introduction of collaborative configuration brings in conflict detection and resolution, which will be studied in Chapter 8.

Finally, in a collaborative environment, workspaces not only manage concurrent commits and updates, they also provide control over who is making a change, when and on which configuration item. These are defined in processes.

3.3.4 Process management

A *process* aims at (1) formally defining the sequence of activities that rule the creation and evolution of software, and (2) guide users during its execution. Process control is widely recognized as a crucial asset of software engineering [PCCW93, tHvdAAR09]. SCM is no exception. An informal study among SCM experts revealed that process support is both the most useful and the most deficient feature of SCM [Est00].

The primary purpose of process support in SCM is to control change requests, bug reports, suggestions for improvement, workspaces, concurrent engineering activities, and configuration. An example of typical change request process is shown in Figure 3.5. A configuration process can be an intertwining of selection of configuration items and of their versions followed by the binding of the artefacts matching an additional set of selection rules. Such a process is sometimes referred to as *multistage configuration process* [CW98].

Research on process support in SCM both explored dedicated mechanisms (e.g. [EDA97]) and the integration of generic process engines. However, preliminary experiments showed that many users could not deal with the flexibility

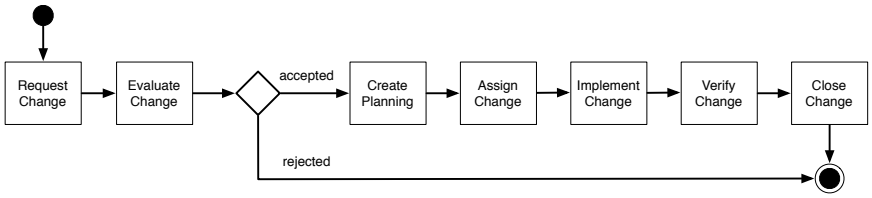


Figure 3.5 – Example of change request process.

of these solutions, and would rather buy and reuse existing processes than develop their own. To reconcile the need for process support and their inherent complexity, tool vendors now hide them behind *“"best practice" standardized processes”* [ELvdH+05].

Process control has now reached its maturity stage in SCM. The challenge ahead is to provide a better integration with third-party tools like call tracking and customer relationship management systems.

FBC barely supports process control. Existing work on that topic will be reviewed in Chapters 6 and 7. We will also propose a combined formalism that maps a multi-view FM to a generic process. We complete the support by providing analyses that ensures that only satisfiable executions of the model are allowed.

3.4 Chapter Summary

Our investigation of configuration in the manufacturing industry suggests that domain-specific interfaces are favoured over generic tree-like representations; no matter what reasoning technique is used in the backend. These observations reinforce our motivation to abstract away from the GUIs for FBC and focus on its foundation. The FM is a raw representation of the options laid out in the interface. We leave it to domain experts to design the interface that is best suited for a particular application.

Another conclusion we draw is that FBC lacks integration with other systems or components (e.g., ERP system or CAD tools). That observation is corroborated by [SK09]. Although commercial tools have started integrating other modelling tools (see Section 2.5), research is still a few steps behind. Our work follows a more holistic approach by placing FBC in a collaborative environment and anchoring it to the engineering process. This approach falls in line with a major conclusion from the systematic analysis of the impact of software engineering research on SCM:

> *[...] the field as a whole is now sorting out how to better fit in*

> *the overall picture of software development, rather than always assuming being able to provide a standalone application that somehow seamlessly fits with existing tools, approaches, and practices.* [ELvdH+05]

The journey through SCM also shed light on the limitations of textual configuration languages, the necessity for process control, the importance of conflict detection and resolution, and the compelling need for workspaces. These three latter observations are supplementary motivations for the work presented in the coming chapters.

Finally, genericity and enhanced expressiveness do not always obtain the preference of final users. Too much flexibility is often seen as a source of confusion, inefficiency, and maintenance overhead.

We are well aware of the limitations of these conclusions. To provide definitive statements, a more systematic and thorough treatment of both AI and SCM is needed but is beyond the scope of this thesis. Yet, the lessons we could learn from our investigation have clearly inspired and influenced our contribution.

Chapter

4 Separation of Concerns in Feature Models

4.1 The Concern Haze

As an FM grows, a recurrent problem emerges: FMs have limited scalability [SCA09]. When describing an SPL with a realistic number of features, typically in their hundreds and even thousands [BSL+10], the models often become too large for human cognitive abilities, too imprecise for automated reasoning, and too overloaded to be purposeful. Evidence for limited scalability of FMs comes from practice (difficulties in creating, updating, and interpreting FMs) [RW06], and from academic research (difficulties in reasoning about large FMs) [SHTB06].

The issue of software design techniques having limited scalability is not new. Various program design techniques, for instance, have had similar limitations. A well-known tactic to manage the scalability problem for program design techniques is *separation of concerns* (SoC). As explained in [Dij82], SoC requires a willingness

> *... to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. [...] But nothing is gained—on the contrary!—by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns" ...*

Therefore, SoC is about recognising that a system may be decomposed using

different criteria [Par72], and the need to be able to distinguish a decomposition made according to a criterion from another. For this reason, there has been much interest in applying the principle of SoC to FMs with the view to making them scale better.

Although the need for separation of concerns in feature modelling is recognised, there is no consensus on what the main concerns of FMs are and how those concerns may be managed. Two issues are addressed in this chapter. First, if separation of concerns is important for FMs, what are the important concerns of FMs to be separated? For instance, FMs may be used to describe design options, choices of user functionality and legal constraints. There are many other legitimate concerns that may be taken into account in FMs. This chapter provides a list of possible concerns for FM languages.

Second, having recognised the concerns, feature modelling techniques need to provide guidelines for how the separation is to be achieved. Separated concerns may also interact and overlap with each other, therefore the question of how to make sense of separated concerns is also important. This chapter provides a discussion of SoC techniques in FM languages.

In addition to these two issues, we will also pay secondary attention to the level of formality involved in the feature language and the (possibility of) tool support for achieving SoC.

Our findings can be used as a basis for (a) providing a multi-concern approach to feature modelling, (b) making FM more scalable through better conceptual clarity, and (c) providing a better formal support for the definition, separation, and composition of concerns.

The rest of the chapter is organised as follows. Section 4.2 discusses the method used to conduct a systematic survey of concerns and their separation techniques in FM languages. Section 4.3 reports on the execution of the paper elicitation process. Section 4.4 summarises seven main threads of concerns and their separation in FM languages. Section 4.5 discusses the findings of the survey and threats to its validity (Section 4.6).

4.2 Survey Method

The research method we have followed to collect and review papers is inspired by the guidelines of Kitchenham *et al.* [Kit04]. Our method deviates from Kitchenham's in that we intentionally leave out a detailed quantitative *meta-analysis* to favour an in-depth qualitative analysis.

This section begins with the presentation of our research questions followed by the description of the survey protocol. It then details the survey material and the data collection forms we used to harvest data systematically.

4.2.1 Research Questions

This survey addresses four questions. The first two questions focus on the elicitation of concerns, their separation and composition. The last two questions evaluate the degree of formality and the tool support that is provided. They are formulated as follows.

RQ4.1 *What are the main concerns of FMs?* We expect there are several FM languages with different notions of "concerns". We will define what is meant by concern in the context of this survey, and list those concerns.

RQ4.2 *How are concerns separated and composed?* Not only that there are different ways of separating concerns, there may be different ways of composing concerns too. The survey will cover those too. In addition, this survey examines how different techniques may have varying degrees of effectiveness.

RQ4.3 *What is the degree of formality?* In order to discover whether separation and composition techniques are amenable to automated processing, we will have to assess the formalism that defines them.

RQ4.4 *Is there (an opportunity for) tool support?*

4.2.2 Survey protocol

The survey protocol is divided into four main steps that go from the selection of papers to their analysis. This process is depicted in Figure 4.1. Starting from the top of the diagram, the survey material is composed of the whole set of papers indexed by DBLP [1] and papers that we know provide answers to the research questions. We detail in Section 4.2.3 how the material was searched, and the papers selected.

The second step is the filtering process during which papers are kept for a complete review. The filtering is based on the search for keywords in the abstract and introduction. In essence, papers that do not refer to feature modelling and separation of concerns in general are discarded.

The third step consists in the complete review of the remaining papers. As advocated in [Kit04], we defined data collection forms, discussed in Section 4.2.4, to systematise this task. The paper review was split between the author of this thesis and another researcher [HTH11]. Our task was to review and fill out the forms for every paper on our list. The final step is a qualitative analysis meant to answer our four research questions.

Our research method combines the completeness of automated search with the reliability of manual reviews. This combined approach helps us keep the

[1] See `http://www.informatik.uni-trier.de/~ley/db/`.

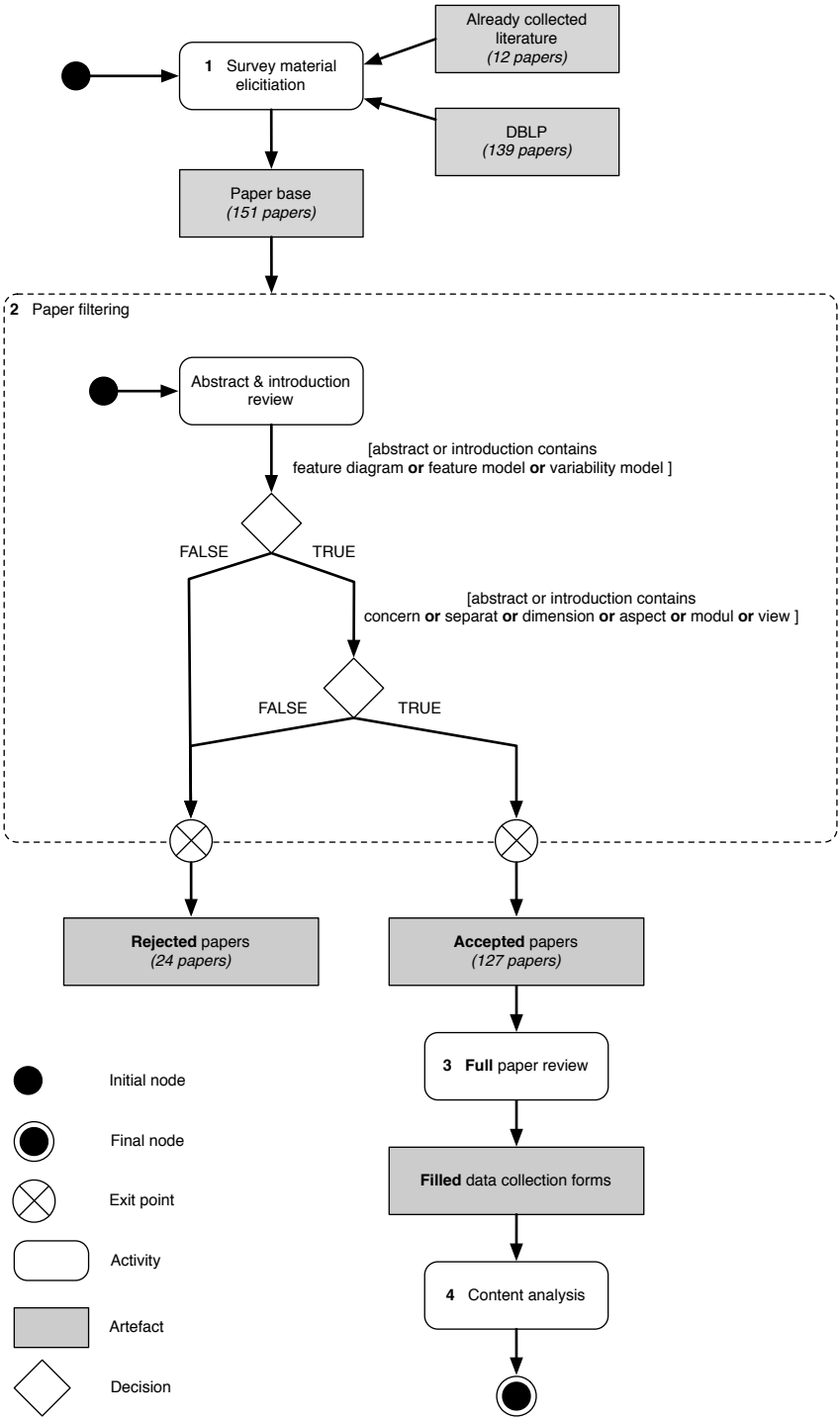


Figure 4.1 – Survey protocol

subjectivity and human effort under control. Reliance on only one of the two techniques can have major weaknesses. For instance, the fully automated search by Chen *et al.* [CAB09] initially missed important publications such as FORM [KKL+98] because the terms did not match.

4.2.3 Survey Material

Our survey covers (1) research papers published in peer-reviewed workshops, conferences and journals, (2) books and other manuscripts, such as technical reports, cited by the peer-reviewed papers, and (3) material accompanying commercial and non-commercial automated tools.

We considered two input sources for our survey: the DBLP computer science bibliography and a collection of papers we amassed in our research on SPL over several years. The search on DBLP was executed through the following five queries:

Q1. `program*.famil*`
Q2. `software*.famil*`
Q3. `product*.famil* concern*|separat*|dimension*|aspect*|modul*|view*`
Q4. `product*.line* concern*|separat*|dimension*|aspect*|modul*|view*`
Q5. `feature*|variabil* concern*|separat*|dimension*|aspect*|modul*|view*`

The `.` makes two strings undividable, meaning that only white spaces can separate the left-hand side from the right-hand side. The `*` is the classical wild card substituting any sequence of non-space characters. Finally, the `|` denotes the boolean *or* and the white space the boolean *and*.

During sample tests of the queries, the version of *Q5* presented above returned more than 600 papers, among which many false positives. Therefore, we restricted the scope of the search to the venues returned by the first four queries. This way, we reduced the number of papers to 69.

4.2.4 Data Collection Forms

[2] These criteria come from [SSS07]. We added the illustration criteria to account for cases where the authors do not perform any systematic validation.

[3] ◧ means that a complete formal semantics is provided. + means that a formal semantics is provided but some aspects are still informally defined. ◯ means that only the abstract syntax is defined by a meta-model or formal grammar. - means that only the concrete syntax is introduced. ▪ means that only informal annotations are proposed.

[4] + means that the activity is fully automated. ◯ means that the automation of the activity is partial and still requires manual processing. - means that no automation is provided.

Research question 1		
Field	**Description**	**Possible values**
Concern	Concerns addressed in the paper	Free form text
Stakeholders	Stakeholders targeted by the concerns	Free form text
Artefacts	Artefacts targeted by the concerns	Free form text
Development stages/tasks	Stages in the software development cycle targeted by the concerns	Free form text
Scope	The scope of the concern determines to what extent the concern is related to the software itself or its environment	*some of* {internal, external}
Target domain	Type of domain to which the concern applies	Free form text
Rationale	Motivation for the need for the concerns	Free form text
Evaluation	Type of evaluation conducted to prove the relevance of the concerns	*some of* {illustration, controlled experiment, case studies, survey research, ethnographies, action research, mixed approach}[2]

Research question 2		
Field	**Description**	**Possible values**
Separation techniques	Techniques used to identify and separate concerns in a FM	Free form text
Composition techniques	Techniques used to compose the concerns of a FM	Free form text

Research question 3		
Field	**Description**	**Possible values**
Formalised artefacts	Artefacts discussed and formalised in the paper	Free form text
Level of formality	Level of formality of the notation used	*one of* { ⊞ , +, ◯ , -, ▪ }[3]
Formalisation	Type of formalisation technique used	Free form text

Research question 4		
Field	**Description**	**Possible values**
Supported activities	Activities supported by the tool	Free form text
Implementation techniques	Techniques used to implement the activities	Free form text
Degree of automation	To what extent the activity is/can be automated	*one of* {+, ◯ , -}[4]
Implementation Status	Level of maturity of the implementation	*one of* {mature, prototype, proof of concept}

Figure 4.2 – Data collection forms

To systematically collect data answering the research questions, we created four data collection forms. To evaluate their ambiguity and completeness prior to the beginning of the actual review, we conducted a pilot test, as recommended in [Kit04]. Five papers were thus randomly selected and reviewed by each of us. We then respectively filled out the forms and reported on their fitness. The adapted forms used for the survey are presented in Figure 4.2. For each field, we provide a short description and the type of expected input.

4.3 Execution

We now report on the execution of the paper elicitation process. First we focus on the results returned by the queries and then discuss the total number of papers sorted by venues.

The five queries were executed on DBLP database on 13 December 2010, using *CompleteSearch* developed by Hannah Bast. [5] 6 shell scripts were used to automatically extract papers and merge the results of the various queries. The five queries respectively returned a total of 144 papers, which are broken down as follows.

Query	**# papers**
Q1	9
Q2	6
Q3	14
Q4	70
Q5	69
Total	144[6]

Of these 144 papers, 122 papers appeared in proceedings and 22 in journals. Table 4.1 synthesizes the distribution of the papers by events and journals. We removed 5 papers that are clearly irrelevant to our survey, and added 12 papers that we know are relevant but whose titles were not matched by the queries. This raised the count of papers to 151. Our qualitative analysis is based on the remaining 151 papers.

From the 151 papers, 15 were not included because they did not match our criteria. 9 more were discarded because they were extended by longer papers. Our qualitative analysis is based on the remaining 127 papers.

[5] `http://dblp.mpi-inf.mpg.de/dblp-mirror/index.php`

[6] There are 24 duplicate papers in the results of the five queries.

Table 4.1 – Distribution of papers by events and journals.

Event	# Papers / Event
SPLC	21
VAMOS	10
ICSE, SAC	7
GPCE, SEKE	6
ICSR, MODELS, OOPSLA	5
AOSD, HICSS	4
APSEC, COMPSAC, ECBS, PFE, RE	3
ARES, DAGSTUHL, ICSOFT, PROLAMAT	2
ACISICIS, CSSE, ECOOPW, EMISA, GCSE, ICAISC, ICECCS, ISPE, ISPW, IWSAPF, MKWI, REFSQ, SERA, SERP, SIGSOFT, UML, VVEIS, WCRE, WICSA	1
Journal	**# Papers / Journal**
TAOSD	5
SIGSOFT, TSE, JUCS, EOR	2
CONCURRENCY, IBIS, IEE, IJMR, IJSEKE, JSS, RE, SCP, SOPR, SPIP, TLSDKCS	1

4.4 Findings

When the data was collected, seven main threads of research began to emerge. They are:

1. *FODA and FODA Extensions.* As the pioneering FM language, FODA and many of its extensions provide a list of concerns for FMs and techniques for their separation. Much of the work in this thread are related to the concepts to be supported by the feature modelling languages.

2. *Aspect-based Concerns.* Although aspect-oriented programming was initially proposed as a mechanism for separating concers in program code, many authors have extended the idea to analytical models of software systems, including FMs.

3. *Requirements Concerns.* Requirements engineering approaches have their own notions of concerns and how to separate them. Many authors have argued that there are requirements-level concerns in FMs, and they can be analysed using techniques developed for requirements engineering. By extension, papers related to problem space, ontologies or natural language also fall in this thread.

4. *Viewpoint-based Concerns.* Originally a requirements engineering approach, viewpoints organise software artefacts according to the stakeholders and their heterogeneous perceptions of the system to build. Separation of concerns is a fundamental issue in viewpoint-based techniques.

5. *Configuration Concerns.* Besides documenting variability, FMs can serve as a backbone for product configuration as they capture all the valid

combinations of features that can lead to software products. Different cross-cutting concerns can intervene during the configuration activity.

6. *Visualisation Concerns.* When FMs are visualised, typically using a tree, there are several cognitive concerns (e.g. number of elements displayed, number of relationships) that need to be addressed, especially when FMs are large and complex.

7. *Architecture Concerns.* Software architecture plays a key role in SPL design approaches, and many of the concerns related to SPL architecture are often reflected in FMs.

We now give a summary of research in each of these area. We note that some of the work fits into more than one of the above categories: therefore, these categories do not represent a firm distinction.

4.4.1 FODA and FODA Extensions

When FODA was first proposed, Kang *et al.* [KCH+90] suggested to divide features into *standard* features, *alternative/optional* features, *specialisation* features, *mutually exclusive* features, and *required* features. From these concepts, it appears that the main concern of FMs is to define valid feature combinations from the perspective of the users.

However, there is more. Another possible concern is to describe feature relationship at binding time: there are *compile-time*, *load-time* and *run-time* features. Furthermore, [KCH+90, KKL+98, LKL02, KDK+02] propose four categories of features. FMs in the *capabilities layer* address functionality as perceived by the end user; FMs in the *operating environments layer* address attributes of the environment in which the application is used; FMs in the *domain technologies* layer address application specific non-technical issues, whilst FMs in the *implementation technologies layer* address technologies that are not specific to a particular domain. These feature categories point to FMs being used to describe feature constraints imposed by several factors, from legal to CPU and to cost.

In addition to these categories, Lee *et al.* use *composed-of*, *generalisation/specialisation* and *implemented-by* relationships [LKL02]. This indicates that FMs also show the refinement relationships between artefacts. The separation technique is *hierarchical layering* of FMs. It is not clear, however, how concerns are managed.

Similar to the layers suggested in [KDK+02], Lee *et al.* provide a further analysis of feature-binding time analysis [LKK04]. In the *product-life-cycle view* there are operation/runtime, preoperation (installation), product development and core asset development stages. In the *feature-binding state view* there are inclusion, availability, activation rule states.

[RW06, RW07, MSA09] consider the issue of managing large FMs and changes made to them over time, in the context of automotive software development. They point out that FMs need to reflect the structure of several organisations involved in the software development. They argue that dividing the FMs along organisational boundaries will make it difficult to propagate changes made to a local diagram. Managing a large global FM is also unsatisfactory because it will make FMs unmanageable. They propose *multi-level feature trees* in which FMs are refined in a hierarchical fashion. Elements of a child FM can selectively reuse elements in the parent FM, allowing local changes to be made without affecting the global structure of the FMs.

Similarly, Thompson *et al.* [TH03] argue, echoing [Lut08], that a single hierarchical decomposition of features is not sufficient to capture the structure of the domain. They propose n-dimensional and hierarchical FMs. They list some of the dimensions as *hardware platform*, *required behaviour*, and *fault tolerance capabilities*. Hartmann *et al.* discuss the issue of FM complexity due to the need to support multiple product lines [HT08]. They propose using a *context variability* model which captures "drivers for variation".

Fey *et al.* suggest that usability and usefulness are important qualities for FMs, and in order to improve those qualities of FODA, they propose a metamodel extending FODA [FFB02]. They argue that feature attributes and complex logical rules are necessary but they are not expressible in FODA. Therefore, they propose to add feature attributes, *pseudo-features* (non-primitive features) and *provided-by* relations to the FM.

Deelstra *et al.* propose an assessment process for evaluating the evolution of variability in software product families [DSB09]. In this approach, a distinction is made between *provided* variability model—the variability in the product family artefacts—and *required* variability model—the variability that is demanded as necessary by product scenarios.

Cho *et al.* identify several feature *relations* and *dependencies* [CLK08]. They are: aggregation relationship, generalisation relationship, required configuration dependency, excluded configuration dependency, usage dependency, modification dependency, exclusive activation dependency, subordinate activation dependency, concurrent activation dependency, and sequential activation dependency.

Kim *et al.* differentiate between *conventional* variability and component variability, and present five types of variability and three kinds of variability scopes [KHC05]. Assuming that classes and workflows among the classes are "*building blocks of software components*", they identify attribute variability, logic (algorithmic) variability, workflow variability, persistent variability, and interface variability. In terms of scope, they propose that a variability may be either binary, selection, or open.

Czarnecki *et al.* propose using probabilistic FMs in the context of mining FMs from existing systems [CSW08]. In this approach, both *hard* constraints

(constraints imposed by the model) and *soft* constraints (conditional probabilities on the selection of features) can be expressed and reasoned about.

There are several other attempts to clarify some of the concerns of FM languages. Etxeberria *et al.* distinguish between *functional* and *quality* features [EM08]. Faulk proposes using *Decision Model* arguing that "*the choices among possible variations must be made to distinguish a family member and any constraints on the ordering of those choices*" [Fau01]. Savolainen *et al.* distinguish between *deductive* and *declarative* properties of product families [SK01].

Summary. FMs were proposed as a way to graphically describe constraints over feature selection within SPL. It has been recognised that these constraints come from several sources, ranging from the end user, the program execution environment, the hardware, the legal bodies, geographical distribution of the software development team, organisational structure, and the refinement of features. However, techniques for separating the concerns, and their composition are rudimentary.

4.4.2 Aspect-based Concerns

Saleh *et al.* discuss application of aspect-oriented programming (AOP) to SPL [SG05]. In particular they discuss how to modularize crosscutting concerns in SPL, where optional and alternative source code is separated from the kernel (common) components. After defining resolution strategies for interacting features, and defining insertion points for the optional and alternative source code, they use code weaving techniques to generate source code for individual products. The concern addressed in this work is the core concern of FMs, namely, separation of core features from optional features.

Similarly, Lee *et al.* call for the need to synthesise feature analysis and AOP because they argue that some features have crosscutting concerns [LKKP06]. Noda *et al.* use aspect-oriented modelling techniques to separate functionalities from crosscutting relationships [NK08]. Coyler *et al.* propose an aspect-oriented approach to separating the concerns of flexibility and configurability of software product lines [CRB04]. They report some heuristics for structuring modules to achieve those qualities.

Batory *et al.* [BLS03] propose an approach for multi-dimensional separation of concerns [TOHS99] in SPLs, which recognises that features may be partitioned in a number of ways (dimensions) and calls the results of each partitioning *units*[7]. For instance, (object-oriented) classification is regarded as a dimension and classes of a software are units. They propose using the *origami matrix* for describing the relationships between units of dimensions. In a simple two-dimensional example, one dimension is for two classes—a singly-

[7] Although this approach is focused on features, rather than FMs, the way concerns are separated is of interest to this survey.

linked list and a doubly-linked list—and another dimension is for additional operations—insert and delete operations. Cartesian combination of the classes and operations gives four possible programs (a two by two table). Units can be *folded* along each dimension: if the operation dimension is folded, there are two available classes, each with insert and delete operations. The class dimension can also be folded in the same way.

Batory *et al.* make two important claims about this approach [BLS03]: (1) it prevents possible invalid combinations of elements; for instance, it does not permit the selection of a doubly-linked list with operations for singly-linked, because folding always has to happen between rows or columns, and not between cells, (2) complexity of n dimensions can be reduced to the complexity of one dimension by folding them.

Elsner *et al.* suggest that model-driven SPL development promises significant benefits [ELSP08]. Model-driven development typically involves distinct steps: from creating computation-independent model to platform-independent model, platform-dependent model to code. In this sense, there is a workflow in the development. They regard features as concerns. Depending on the feature selection, the workflow required to instantiate the required product may vary. They also argue that current workflow languages are not adequate in expressing feature-based modularisation of software systems. They propose an aspect-oriented approach to separate out the base workflow from additional workflows, which can be weaved into the base workflow when additional features are selected.

Summary. Aspect-orientation seems to provide a mechanism for separating core software artefacts from optional ones, and weave them through well-known aspect-weaving techniques. This synergy between FMs, SPL and aspect-orientation has been exploited by several SPL approaches [BM09].

4.4.3 Requirements Concerns

Pohl *et al.* differentiate between *variability in time*—denoting changes to artefacts over time—and *variability in space*—denoting static variability of artefacts [PBvdL05]. They further distinguish between *external variability*—relevant to customers— and *internal variability*—relevant to developers.

Similarly, Metzger *et al.* [MHP+07] propose distinguishing two kinds of variability, *product line variability* and *software variability*, where the former is concerned with the "*ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context*" [SvGB05], whilst the latter is concerned with "*the variation between the systems that belong to an SPL in terms of properties and qualities, like features that are provided or requirements that are fulfilled*".

Classen *et al.* disambiguate the meaning of a feature in requirements engineering [CHS08]. They build upon the work of Jackson, Zave and others

[Jac95, ZJ97, GGJZ00] to give a precise definition of a feature as triple composed of *requirement* (R), *specification* (S) and *domain assumptions* (W) such that S and W entail R. They also revisit how that extended definition affects the notion of *feature interaction.* Along the same line, Tun *et al.* propose to follow the Jackson–Zave framework to separate the concerns of FMs [TBC+09]. In addition, they express quantitative constraints on the feature modes, links connecting the three models, in order to generate feature configurations that satisfy stated requirements and quantitative constraints.

Moreira *et al.* argue that separation of concerns is not two-dimensional, as assumed in the distinction between functional and non-functional requirements (NFR) where NFR crosscut the *base* functional decomposition, but are multi-dimensional where both functional and non-functional requirements crosscut each other [MRA05]. They divide concerns into *meta concerns*—generic to many systems—and *system concerns*—particular to a given system.

Faulk distinguishes *domain engineering* from *application engineering* where requirements documents produced in the former phase are reused in the latter phase [Fau01]. Grübacher*et al.* discuss the challenges of structuring the modelling space for SPLs [GRDL09]. They argue that maintaining a single FM for the entire system is not feasible and proceed to suggest strategies for feature modelling from various perspectives. They also present some examples of how these strategies can be applied and supported by existing tools.

Czarnecki *et al.* describe FMs as views on ontologies, because they consider ontologies to be more expressive than FMs [CKK06]. Lauenroth *et al.* describe a meta-model of variability in software product lines [LP08]. However, they do not discuss what concerns might be represented in the variability models.

Hallsteinsen *et al.* describe an approach to developing adaptive systems where they separate concerns of functionality from that of adaptivity [HSSF06]. Kircher *et al.* report on the importance of the separation of *problem space* from the *solution space* where, for instance, the distinction of user requirements from technical requirements is emphasised [KSG06].

Breen describes some experience of developing SPL [Bre05]; one of the issues of developing behavioural specifications of features is scalability. Individual features are specified in tabular format and they are synchronised through the use of internal and external events in the specifications. Waldmann *et al.* report on the industrial experience of defining an SPL based on features [WJ09].

Niu *et al.* use text processing techniques to extract variability models from natural language requirement statements [NE08]. Aoyama proposes using *persona* to elicit requirements where users are divided into groups according to *segmentation variables* such as age, gender and occupation [Aoy05]. Requirements are divided into services offered by the system, and the services are then matched with persona.

Assuming that features are concerns, Heidenreich *et al.* compare two approaches to concern composition: a declarative approach (FeatureMapper) and

an operational approach (VML*) [HSS+10] . They report that the two approaches are equally powerful on important criteria.

Chen *et al.* divide requirements into candidate features for *crosscutting* requirements and identify two types of feature dependencies, called interaction and weaving [CZZM06]. In [CZZM05], they propose a way of constructing FMs based on requirements relationship graphs and a clustering technique.

Summary. Many requirements engineering approaches tend to focus on separating the user's perspective on a FM from the perspectives of other stakeholders. There are also approaches that differentiate between features relating to functional and non-functional requirements, to different personas, and to phases of development.

4.4.4 Viewpoint-based Concerns

Choi *et al.* divide FMs according to five views [CLLK09]. By views, they mean ways to characterise extensions to FM languages as opposed to describing what the main concerns of FMs are. In the *structure view*, feature relationships such as aggregation and generalisation are described; In the *configuration view*, mandatory, optional and alternative features, together with feature cardinality, and group cardinality are described; In the *binding view*, feature binding units representing groups of features bound together are described; In the *operational dependency view* the interactions between features are addressed through the identification of dynamic dependencies among them; In the *traceability view*, the implemented-by relationships between features are shown.

In [HHB08], we investigate the practical challenges of applying FM languages. The main challenge we report is that of making modelling perspectives (such as design time versus runtime perspectives) explicit in FMs.

Summary. Views on FMs very widely. They cover structural (constructs of the language), operational (binding time and dependencies), and contextual (relationship to external artefacts and variations in the runtime environment) viewpoints.

4.4.5 Configuration Concerns

Czarnecki *et al.* propose *staged configuration* approaches FMs are specialised in a stepwise fashion, and instantiated according to the stakeholder interests at each development stage [CHE04, CHE05]. With *specialisation* they refer to a process in which variability in FMs is removed. In other words, a more specialised FM has fewer variabilities than its parent FM. A fully specialised FM has no variability. A *configuration*, on the other hand, is an *instantiation* of an FM. Staged configuration has been formalised in the dynamic semantics of FMs by [CHH09b].

With *multi-level* staged configuration, Czarnecki *et al.* refer to a sequential process in which an FM is configured and specialised alternately by stakeholders in the development stages [CHE05]. For instance, a stakeholder will instantiate an FM by selecting features that are relevant to its requirements. The instance of the model is then used to specialise the FM by removing parts of the model that are no longer available. The resulting FM is then instantiated by another stakeholder, and so the process repeats itself.

In [HHSD10], we build upon multi-level staged configuration and propose a combined formalism that maps views on a FM to tasks in a workflow that models the configuration process [HCH09]. We thus achieve better separation of concerns between the configuration process and the model. Unlike multi-level staged configuration, we also allow complex processes to be modelled. In the same vein, White *et al.* describes a model of configuration steps [WBDS09]. Mendonça *et al.* present algorithms that aggregate features into coherent groups and then computes possible configurations sequences for these groups of features [MCMdO08, MBC08].

Summary. Concerns in these approaches are similar to those of viewpoints, but rather focus on the way the FM can be chunked up, the configuration of the FM, and the processes that pilot the configuration.

4.4.6 Visualisation Concerns

[BTN+08, CTH08] discuss how visualisation techniques can help explore complex FMs. First they propose a meta-model that covers three models, namely, *feature model*, *component model*, and *decisions models* to provide ways to navigate the FM. The tools they propose allow *details on demand*, *incremental browsing*, *focus+context*, and ways to visualise product configuration. *Visual concerns* of FMs are the ability to relate these three dimensions.

Gonzalez *et al.* propose a tool-supported visual analysis approach to examine the relationship between non-functional softgoals and functional goals for requirements variability [GBPLM04]. Ribeiro *et al.* address the issue of conflicts in Virtual Separation of Concerns where developers work on different code segments which overlap [RPTB10]. They propose defining contracts between features, which are used to prevent developers from performing conflicting edits.

Summary. The visualisation concerns of FMs abstract FMs according some given criteria and render them visually.

4.4.7 Architecture Concerns

Liu *et al.* propose a kind of connector called *concern connector* to define the scope of concerns in architectural design and use it to evolve product line software where safety is a critical concern [LLT05]. The authors argue that

the notion of "hyperspace" provides a framework for separating concerns, and concern connectors provide an architectural concept to reconnect the separated concerns.

Thiel *et al.* propose four basic extensions to IEEE Architecture Descriptions to accommodate product line software [TH02]. The SPL specific challenges they have identified are: *product line extension* to capture the relationships between a product line, product, and product line architecture; *feature variability extension* to relate variability to the requirements; *architecture variability extension* to relate variability to architectures; and *design element extension* to relate the component and connection types to the architectural variability. Olszak *et al.* present a semi-automated method analysing execution traces of object-oriented programs in order to locate features in the programs [OJ09].

van Zyl describes a two-dimensional "continuum" of concerns [vZ02]. The horizontal continuum reflects the software layers whilst the vertical continuum reflects the development artefacts. In the *horizontal continuum*, there are two extremes: *client facing aspects* and *infrastructure facing aspects*, and also layers similar to those in the Model-View-Controller architecture. In the *vertical continuum*, there are: *business model*, *system model*, *application portfolio*, *service architecture*, *component architecture*, and *object architecture*.

Krueger illustrates an application of the *divide and conquer* approach to identifying subproblems in variation management [Kru02]. He uses a two-dimensional table, where three rows represent *granularity of software artifacts* (Files, Components, and Products) while the three columns represent *variation types* (Sequential Time, Parallel Time and Domain Space). Each cell in the table gives a subproblem to solve.

Niemelä *et al.* provide an approach to capture quality requirements and transform them into architectural models [NI07]. Zhang *et al.* use feature dependency relationships to analyse requirements and relate them to design artefacts [ZMZ06].

Wohlstadter *et al.* examine the issue of managing runtime variabilities introduced by the execution environments of the programs [WD06]. They propose an interface description language and a mechanism for co-ordinated adaptation of distributed components to specific execution environments.

Summary. The architectural concerns relate generally to concrete domain artefacts. The FM is usually one element of the set of concerns that characterise the architecture of the SPL.

4.4.8 Overall Findings

Our findings show that there are numerous ideas for what should be the main concerns of FM languages. Although, a consensus is yet to emerge, design and requirements concerns appear to be the dominant concerns of FMs. Other types of concerns appear to be secondary. Given the relatively long history of the SPL

literature, it is somewhat surprising that FM languages have not formulated their concerns precisely. This could be a consequence of the difficulty to thread together all the papers, which barely build on each other. The fragmentation and lack of cohesion of the SPL literature is the main problem we faced.

In terms of separation techniques, aspect-based approaches have been proposed as a possible technique for managing concerns in FM. Although the successful use of aspect-based techniques in programming has been acknowledged, their application to FM languages has yet to achieve the same level of success.

Although there are some notable exceptions, the lack of formality and tool-support is in general rather glaring in the SPL literature.

4.5 Discussion

When FMs were first introduced, the main concern of the models seemed to be to distinguish features that are particular to certain members of the product family from those that are common across all members. FMs address something that few other modelling languages do: to treat groups of programs as having common functionality with only minor varying features. This could encourage reuse across programs, and design more stable architecture. The concepts used in the language were also simple: mostly with AND (aggregation/consist-of) and OR (alternatives/optional/mutually exclusive OR) relationships, together with some notion of feature attributes and "requires" (implies) relationship [KCH+90].

Having said that, even the seminal FODA report [KCH+90] is not clear about the level of abstraction of features, nor the purpose of FMs. At times, [KCH+90] treat FMs as a *domain analysis* tool, suggesting that the feature analysis should focus on application domain; at other times, they indicate that FMs can be used to describe relationships between features of different binding times: compile-time, load-time and run-time. Therefore, an FM is a "*communication medium between users and developers*", a description of the "*user's understanding of the general capabilities of applications in a domain*", and a tool for communication between developers and designers. This vagueness in the purpose of FM language, or the attempt to cater for all purposes, unfortunately contributes to FMs with poor separation of concerns.

This section revisits the findings of Section 4.4. Firstly, it synthesizes the different types of concerns we identified and tries to clarify their meaning and context of use. Then, it elaborates on the different separation and composition techniques. Finally, the degree of formality of these techniques and the tools supporting them are discussed.

4.5.1 Concerns

Two types of concerns are proposed in the literature: (1) concerns that group features together according to some given criteria; (2) concerns that differentiate the kinds of relationships between features.

Feature groups

Concerns grouping features can be further separated into three categories, as reported in Table 4.2. First, concerns that focus on *functional and non functional properties* of the SPL. Then, concerns that put together features that address a specific *facet* of the SPL variability. Finally, concerns that group features according to the *steps or status of the configuration process*.

It is interesting to note that the earlier FM languages (such as FODA and FORM [LKL02, KKL+98]) seem to be more concerned with design and implementation issues, while later FM languages (such as staged configuration [CHE05] and feature tree [RW06]) are more concerned with stakeholders and organisational structures. There is a tendency to expand the scope of FMs: in addition to describing variability in the design, a need for describing the variability in the wider system context has been recognised by SPL approaches. This perhaps explains, in part, the phenomenon of increasing size and complexity of FMs.

A modelling language such as the one described in FODA cannot be expected to adequately capture relationships between so many (kinds of) features in a realistic system at so many levels of abstractions, involving so many stakeholders at the same time. For example, there is no notion of timing (to talk about temporal dependencies between features) in FMs, although FODA authors suggested that they can be used to describe run-time and load-time variabilities.

In that sense, it is not a real surprise that many FODA-like FM languages are found not to be scalable. This has led some authors to question the actual expressiveness of those languages. As shown in [SHTB07], from the semantic perspective, a relatively simple FM language can have a high degree of expressiveness and succinctness. Therefore, we have to conclude that making FM languages richer does not necessarily make them more scalable. Poor separation of concerns makes FMs difficult to understand and analyse using automated tools.

The list of concerns recognised by the surveyed approaches, in particular by [LKL02] and [KKL+98], is very comprehensive. They range from cost of features, CPU platform to organisational structure. This indicates that variability has to be addressed at different times in the development and in different parts of the system structure. Therefore, we do not have a shortage of concerns to be addressed by FMs: there are actually too many concerns to be addressed.

Table 4.2 – Concerns separating groups of features

References	Concern
Functional and non-functional property	
[HSSF06]	Adaptivity
[CRB04]	Configurability
[SK01]	Declarative
[SK01]	Deductive
[TH03]	Fault tolerance
[CRB04]	Flexibility
[EM08, HSSF06]	Functionality
[EM08]	Quality
Facet	
[SG05, ELSP08]	Artefacts
[CLLK09]	Binding
[KCH+90, KKL+98, LKL02, KDK+02]	Capability
[CLLK09]	Configuration
[KCH+90, KKL+98, LKL02, KDK+02, CHS08, HT08, TBC+09, MRA05]	Context
[KCH+90, KKL+98, LKL02, KDK+02]	Domain technology
[PBvdL05]	External variability
[CZZM06]	Feature interaction
[TH03]	Hardware platform
[KCH+90, KKL+98, LKL02, KDK+02]	Implementation techniques
[PBvdL05]	Internal variability
[GRDL09]	Market needs
[HT08, GRDL09]	Multiple product lines
[FFB02, SHTB06]	Non-primitive feature
[CLLK09]	Operational dependency
[RW06, RW07, MSA09, GRDL09]	Organisational structure
[DSB09, MHP+07, MRA05, KSG06, CHS08, TBC+09, GRDL09]	Provided variability (solution space)
[DSB09, MHP+07, KSG06, TBC+09, TH03, CHS08]	Required variability (problem space)
[CLLK09]	Structure
[CLLK09]	Traceability
[PBvdL05]	Variability in time
[PBvdL05]	Variability in space
Configuration process	
[KCH+90, LKL02, KDK+02, LKK04, FFB02, HHB08]	Binding time
[LKK04]	Binding state
[CHE05, CHH09b]	Configuration levels
[CHE04]	Configuration stages
[HCH09]	Configuration tasks

This is one of the fundamental problems of FMs. Again, a clarification of the purpose of FMs, and the meaning of features in various artefacts, will go some

way to address the problem.

In addition, several approaches group features according to the artefacts of which they model the variability. Some of them directly embed variability into artefacts such as UML models [ZHJ04, GS08, HSS+10]. These approaches do not consider variability models as first-class citizens. Conversely, other approaches handle the variability model and base model separately. For instance, orthogonal variability modelling (OVM) [PBvdL05] models variation points individually and links them to base models, Czarnecki *et al.* [CHE05, CP06] separate FMs from the base UML model, Classen *et al.* formally relate an FM to a behavioural model and reason about them both [CHS+10], and Heidenreich *et al.* map FMs to other models such as use cases, class diagrams or statecharts [HSS+10]. Our work focuses only on approaches that recognize FMs as first class citizens. In that context, a concern is defined as the mapping between the FM and a set of models. However, most of the work in that field falls outside our research questions. Therefore, we do not try to provide an exhaustive list of them here.

Feature relationships

The second type of concern determines the nature of the relationship between two or more features. Table 4.3 collects the different kinds of relationships we came across. Relationships defined as propositional formulas [Bat05] that are not explicitly qualified are left out this table.

Here again the concept of relationship is rather imprecise. It is alternatively used to specify the parent/child relationship (e.g., aggregation or composition), to constraint feature selection (e.g., requires or excludes) and to provide dependency information (e.g., implemented by or usage). Many of them defeat the purpose of FMs. By forcing implementation specific details into the model, one loses independence from the solution domain. Furthermore, the semantics of these relationships is for the most part completely overlooked.

However, implementation-specific relationships differ from implementation specific feature groupings. Relationships restrict the selection by enforcing constraints imposed by technological decisions. Groups present domain options that are related to each other. Even though concerns can focus on implementation-specific features, they only expose choices and do not import extraneous constraints from the implementation layer.

4.5.2 Separation and composition techniques

In the same way that we have many possible concerns of FMs, there are also many ways to separate concerns including: aspect-oriented, view-oriented, tabular, visualisation, staged-configuration, requirements-oriented approaches. Having a diversity of approaches in itself is an advantage. However, since the

Table 4.3 – Concerns separating relationship among features

References	Concern
[CLK08]	Aggregation relationship
[LKL02]	Composed-of
[CLK08]	Concurrent activation dependency
[CLK08]	Excluded configuration dependency
[CLK08]	Exclusive activation dependency
[CHS08, TBC+09]	Entailment
[LKL02, CLK08]	Generalisation/specialisation
[CSW08]	Hard constraint
[LKL02]	Implemented by
[CZZM06]	Interaction
[CLK08]	Modification dependency
[WBDS09]	Point configuration constraint
[FFB02]	Provided by
[TBC+09]	Quantitative constraints
[CLK08]	Required configuration dependency
[CLK08]	Sequential activation dependency
[CSW08]	Soft constraint
[CLK08]	Subordinate activation dependency
[CLK08]	Usage dependency
[CZZM06]	Weaving
[MHP+07]	X-links

purposes and concerns of FMs are not clear, the concerns to which these techniques are to be applied remain a question. Many of the proposed approaches are well-grounded and probably constructive when applied to real problems. More evidence of how they have been applied will strengthen confidence in them.

Although there are several separation techniques, as shown in Table 4.4, what is required, in our view, is a clarification in the purposes of FMs, a systematic way to separate, and compose FMs, according to well-defined criteria. The Jackson-Zave framework (and related) for requirements engineering gives a systematic way to separate requirements, specifications and domain properties. We believe that FM languages could benefit such a framework, as explained in [CHS08, TBC+09].

Separation of concerns is only one side of a two-sided story: having separated concerns, there is a need to reason about how concerns might be related and negotiated. Many of the techniques are summarised in Table 4.5. Regarding this issue, we see a deep synergy between SPL and requirements engineering research. For instance, techniques on viewpoints [EN96], model synthesis [UC04], model merging [BCE+06, ACLF09], inconsistency management [SZ01] may provide insight into how concerns on FMs could be managed.

Table 4.4 – Separation techniques

References	Separation technique
[SG05, LKKP06, NK08, CRB04, ELSP08]	Aspect
[MCMdO08]	Configuration spaces
[HHSD10]	Configuration views
[KCH+90, KKL+98, LKL02, KDK+02, TH03, RW06, RW07, MSA09]	Hierarchical layering
[CHE05, CHH09b]	Level
[TH03]	n-Dimensions
[BLS03]	Origami matrix
[CHE05, CHH09b]	Stage
[WBDS09]	Step
[HCH09]	Task

Table 4.5 – Composition techniques

References	Composition technique
[BLS03]	Cartesian combination
[MCMdO08]	Configuration plan
[MHP+07, TBC+09]	Crosscutting constraints
[HCH09]	Feature configuration workflow
[RW06, RW07, MSA09]	Resolution rules
[CHE05]	Specialisation
[SG05, CZZM06]	Weaving
[ELSP08]	Workflow weaving

4.5.3 Formalisation

In terms of level of formality, most of the FM languages we surveyed are largely informal. We believe that the question of having an expressive and succinct formal FM language is no longer a major problem. The major problem is that most of the approaches to separation and composition of concerns are weakly formalised. Only 19 out of the 127 papers obtained a score of ◼ or +, which means that a formal semantics is provided. 10 other papers obtained ◯ because they provide an abstract syntax or metamodel of the concepts they use. Some of the papers we reviewed formalise concepts that are not directly related to FMs (e.g. [AC06]). In general, what we observed is an informal and loose definition of the concept of concern.

On top of that, the very nature of a concern varies. Besides the distinction between group concerns and relationships, more fundamental differences can be noticed. For group concerns, some seem to be "attributes" characterizing features like the binding time. Others make a clear separation between FMs achieving different objectives like provided vs. required variability. Finally, a concern can be captured by a link to an external model. The same observation holds for relationship concerns that can either denote a specific type of

constraint or represent a form of conceptual consistency like *entailment*.

There is apparently no determination to reach a unified definition, which prevents a rigorous comparison of the approaches. To systematise the way the problem is tackled and converge on a solution, the following questions should be answered:

- *What are the types and subtypes of concerns?* Different types of concern exist. Tables 4.2 and 4.3 lay the foundation stone by collecting two of them, i.e. feature *groups* and feature decomposition *relationships*. Future work should tell which more fine grained types are needed, and categorize them more precisely. However, a more pressing matter might be to produce solid empirical evidence that these concerns are actually relevant in practice. Therefore, further investigation is needed to populate a repertoire of relevant concerns.

- *How is a concern defined?* Several papers suggested formal definitions of a concern (e.g. [BLS03, RW06, HCH09, MCMdO08, HHSD10]). However, these definitions focus on feature groups with simple parent/child relationships and boolean constraints. Efforts are still needed to integrate them and extend them to account for the different types of concerns (e.g. using an attribute vs. using a link to an external model than contains equivalent information).

4.5.4 Tool support

Many of the tools we came across are little more than simple diagramming tools or standard aspect code-weaving tools. For the most part, they are only prototypes or proofs of concept. Less than five of these tools seem to have reached a mature stage. Meaningful and efficient reasoning about multi-concern FMs is still an area for further research. For instance, how separation of concerns, and their recomposition should be supported by an automated tool is under-explored. This is a likely consequence of the limited research on concern formalisation. Better specification of the problem would lead to more robust and efficient tool support.

Despite the apparent difficulties, SPL engineers in various industries have been successfully producing commercial software used by many customers. Insightful reports on how SPL engineers actually manage concerns in FMs would have a positive influence on research. How tools like pure::variants [Beu08] that allow to add numerous attributes to a feature can serve as a sound basis for concern modelling is still unclear.

4.6 Threats to Validity

The last step of our survey is a critical look at the research method and the collected results. For each point, the threats to validity are discussed with the presumed consequences on the global validity of the results.

Single bibliographical database The only bibliographical database we used to collect papers is DBLP. The bias that could have been introduced was however mitigated by (1) the automatic inclusion in DBLP of papers from other electronic libraries (ACM, IEEE, Springer...), and (2) the list of papers we manually added. The latter reduced the risk to miss relevant papers not caught by the queries or not encoded in DBLP.

No meta-analysis We favoured a qualitative approach over a quantitative one. Consequently, we neither completed a meta-analysis nor a sensitivity analysis. A meta-analysis is usually meaningful to aggregate results from various yet similar studies [BKB+07]. The variability among the papers we collected simply rendered impossible such aggregation. It turned out that tabulated data [BKB+07] was the only way for us to compare the results. This, in turn, made a sensitivity analysis inappropriate.

No external expert We did not appoint an external expert to control the completeness and consistency of the review. The impact it has had on the final results is hard to tell given the broad scope of our investigation. We tried to lessen the possible negative impact by appointing two different researchers with some overlap in our assignments to make sure that our evaluations were consistent and comparable.

No quality evaluation To build a complete repertoire of concerns, all papers were equally evaluated without any discrimination regarding the source (e.g. journal, conference or workshop) or venue. A quality evaluation would have excluded several papers that contained valuable input for our repertoire. To reach that level of exhaustiveness, quantity was preferred over quality. The overall impact on the quality of study, however, is hard to predict. In fact, the fairly low level of empirical validation in all the papers lead us to believe that no significant gains in quality would have been perceived if we had been more selective on venues.

Inconsistent content validation We observed significant differences in the validation of the results presented in the papers. For instance, many papers claimed that their results were validated with case studies. Yet, few of them, followed a rigorous approach to empirically validate their results such as [SSS07]. That threat affects research on separation of concerns in FMs in general. Our study can only reflect the current state of the art in that domain.

4.7 Chapter Summary

This chapter exposed the results of a systematic survey of literature on separation of concerns in FM languages. Our research method is well-founded and covers both quantitative and qualitative aspects. The main research question of the survey has been to identify the most important concerns, or purposes, of FMs, and how they are separated in existing SPL approaches. The keys findings are the following:

- We have identified seven areas of research in which concerns of FMs are discussed: the concerns range from the legal constraints on the feature combination to the hardware and organisation structure constraints.
- The inherent vagueness in the feature abstraction and in the purpose of FMs makes realistic FMs hard to comprehend and analyse. Since a more expressive FM language is unlikely to solve the problem, we have concluded that a clarification in the purpose of FMs, and the meaning of features in various artefacts will go some way towards solving this problem.
- Separation and composition techniques for concerns in FMs have been found to be rudimentary: they range from hierarchical layering of FMs to selective projection of FMs using a visualisation tool.
- Tool support for aspect-oriented approaches seems to offer some opportunity, but so far they have been mainly applied to code, rather than FMs. We have concluded that a firmer conceptual footing for FMs will enhance tool support for separating and composing concerns.

There is still a long way to go before a complete list of concerns can be provided. Additional work is needed to comprehend the relationship between FMs and other types of models. Besides bringing in new concerns, that will require dealing with many extra separation and compositions techniques mixing features and heterogeneous model elements.

Rather than studying each type of concern individually, the following chapters investigate the specification, verification and rendering of concerns with generic views (Chapter 5); define a formal way of driving view configuration with a workflow (Chapter 6 and 7); propose alternative solutions to handle conflicts during the concurrent configuration of multiple views (Chapter 8); introduce a toolset implementing those techniques (Chapter 9).

Chapter

5 Multi-view Feature Models

5.1 Open Issues

Two challenges that FBC techniques fail to address in a satisfactory way are (1) *tailoring the configuration environment* according to the stakeholder's profile (knowledge, role, preferences...), and (2) *managing the complexity* resulting from the size of the FM. Two concurrent approaches have been proposed to tackle that problem: *view integration* and *view projection*. View integration techniques start from small and roughly independent FMs that are configured and then integrated to form a complete configuration (e.g. [CKK06, RW06, MSA09]). In practice, the gain of designing and working with smaller models often echoes with costly and complex integrations of heterogeneous models [GRDL09]. Conversely, view projection techniques assume the existence of a global FM that is divided into smaller views, which are then configured. The high upfront investments necessary to build the initial FM is counterbalanced by the automatic integration of user decisions. The wide application of that latter approach in various domains (see e.g. Chapter 4, database engineering [EN06], and product line implementation [KÏ0]) and our own experience showing that a single feature model is usually favoured over an heterogeneous collection thereof lead us to focus on view projection.

In FBC, a view is a simplified representation of an FM that has been tailored for a specific stakeholder, role, task, or, to generalize, a particular combination of these elements, which we call a *concern*. Views facilitate configuration in that they only focus on those parts of the FM that are relevant for a given concern. Using multiple views is thus a way to achieve SoC in FMs. SoC helps making FM-related tasks less complex by letting stakeholders concentrate on

the parts of the FM that are relevant to them and hiding the others.

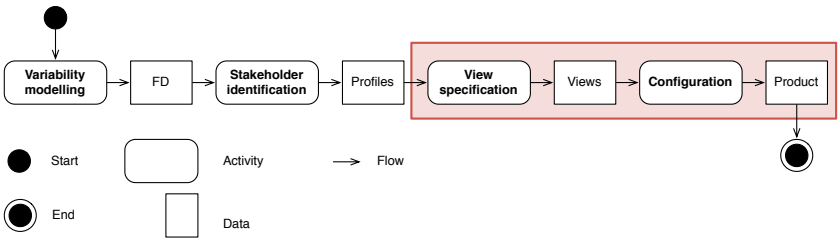


Figure 5.1 – Key steps and artefacts of FBC.

In Figure 5.1, we represent the key steps and artefacts of FBC that use view projection. Typically, the process starts with variability modelling, which produces the FM. The second step focuses on the identification of stakeholders, which determines the profiles of the users of the configurator. Then, a view on the FM is defined for every profile. The final step is the actual configuration of the FM, which results in the specification of the product.

In this chapter, we focus on the creation of consistent views and the generation of alternative visualisations for configuration, as highlighted in Figure 5.1. Specifically, we address three fundamental issues of multi-perspective feature modelling:

RQ5.1 *How are views actually specified?* Related work usually identifies views by surrounding groups of features with a line or by showing subsets of features from the original FM. This gives very little insight as to how to actually build these views in practice and certainly does not tell how to implement a tool that renders them.

RQ5.2 *How is the complete configuration of the FM enforced?* Views delimit portions on the set of features. To be meaningful, views should cover the complete decision space defined by features, i.e., one should ensure that no feature of the FM can be left undecided.

RQ5.3 *How are features outside a view filtered out?* Some stakeholders need to see features outside a view to comprehend it. However, for large or technical FMs, the complexity can become disorienting and features outside the view have to be hidden to simplify the configuration task. Finally, security policies can restrict the set of stakeholders who can access (read/write) particular features. These different scenarios put different constraints on view rendering.

Our contribution is a set of techniques to specify, automatically generate and check multiple views (Section 5.3). Views are generated through trans-

formations of the FM. Verifications and transformations are formally defined (Section 5.3), and applied to the PloneMeeting case (Section 5.4). The correctness of the transformations is finally demonstrated (Section 5.5). But first, let us introduce our working example: PloneMeeting.

5.2 Working Example: PloneMeeting

PloneGov[1] is an international open source initiative coordinating the development of secure, collaborative and evolutive eGovernment web applications. PloneGov gathers hundreds of public organizations worldwide. This context yields a significant diversity, which is the source of ubiquitous variability in the applications.

Our collaboration with PloneGov developers aims at addressing those variability management challenges [DMH+07, HHB08, UDH09]. We concentrate here on PloneMeeting, PloneGov's meeting management project, which has now been re-engineered as an SPL. A major challenge was to extend its flexibility through systematic variability management. We collaborated with the developers in designing an FM representing the configuration options of PloneMeeting. A sample of this FM [2] is presented in Figure 5.2. The extra constraints appear in the lower right corner. The coloured areas should be ignored for now. The essential concepts of this model are introduced below.

Meeting management typically follows a three-step process: (1) meeting items, i.e. points to be discussed, are created and validated; (2) a meeting is created and existing meeting items are put on its agenda; (3) after publication, the meeting takes place and the decisions made on items are archived. In PloneMeeting, each item and meeting has its own statemachine, reflecting the management workflow. A typical workflow contains states like "Created", "Closed" or "Archived". The states and transitions of the workflow are selected and possibly customised during the installation of PloneMeeting to be compliant with local policies. PloneMeeting also provides support for basic task management and electronic voting.

PloneMeeting recognizes three different stakeholder profiles. Each of these profiles independently configures a part of the website. The *web administrator* is a Plone expert in charge of the installation, maintenance and update of the PloneMeeting instance. The *PloneMeeting manager* is responsible for the base configuration of the website, including meeting workflow definition. The *users* directly exploit the meeting management functionalities as participants, meeting managers, observers, etc. The configuration options of interest for each of these profiles are thus different and limited in scope.

[1] `http://www.plonegov.org/`

[2] Reverse-engineered from PloneMeeting version 1.7 build 564.

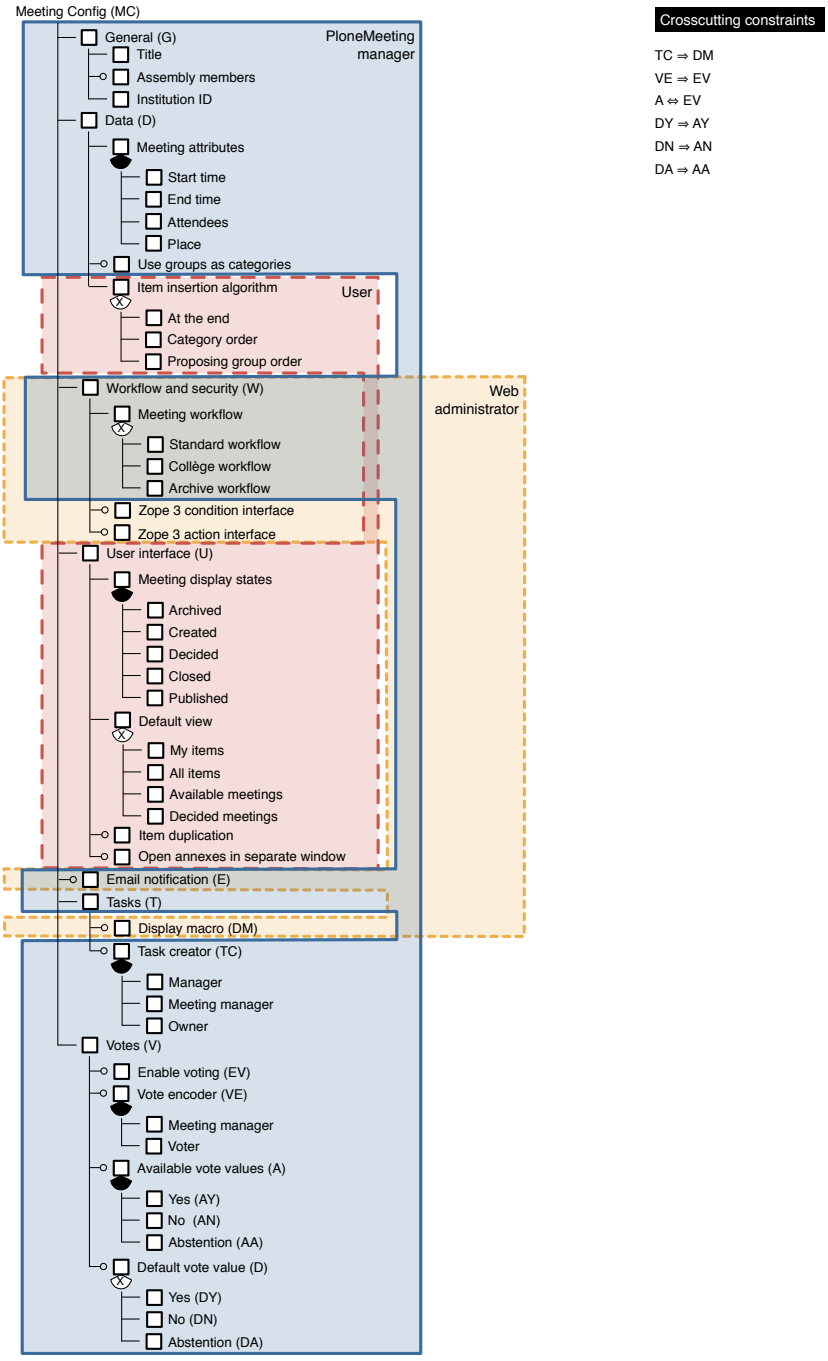


Figure 5.2 – Excerpt of PloneMeeting's FM.

A major problem is that existing FBC tools do not provide means to control who is responsible for a given configuration option. At best, informal comments or notes can be added to tag features, which severely limits the automated tailoring of configuration interfaces and the control over access rights. However, these two functionalities are required by PloneMeeting. In the absence of clear access specifications, a coarse policy has been implemented: the web administrator and the PloneMeeting manager have both access to all configuration options, while the users get access to none. A reported consequence is that sometimes the PloneMeeting manager does not have sufficient knowledge to fully understand the options and make decisions. The results can be incorrect settings of interfaces to external macros and runtime changes of meeting workflows that lead to inconsistent meeting states. Additionally, users are denied any tailoring of their working environment, e.g., default GUI layouts or choosing how to display states of meetings or other items.

Furthermore, responsibilities and profiles can vary from one PloneMeeting instance to the other. The variability in the use context might imply variations in the access rights (e.g. the PloneMeeting manager cannot control workflows). It might also require other stakeholder profiles (e.g. a *Task Manager* is needed to configure the task portlet). Consequently, responsibilities and profiles should not be hardcoded in the FM but defined on a case-by-case basis, typically before or during the instantiation of the website.

This situation provided the initial motivation for the use of views as a flexible means to build and reason about configuration spaces. However, existing solutions to multi-view feature modelling fail to provide complete support to model this case.

5.3 View Definition, Verification and Visualisation

5.3.1 Basic definition

Answering research questions *RQ5.1-3* requires being able to specify which parts of the FM are configurable by whom. This can be achieved easily by augmenting the FM with a set V of views, each of which consists of a set of features. Based on the definition introduced in Section 2.3.2, a *multi-view FM* is finally defined as follows.

Definition 5.1 **Multi-view FM ($\mathcal{L}_{MVFM}$)**

A multi-view FM $u \in \mathcal{L}_{MVFM}$ *is a tuple* $(N, P, r, \lambda, DE, \Phi, V)$ *where* $V = \{v_1, v_2, \ldots, v_n\}$ *is the multiset of views such that:*

- $N, P, r, \lambda, DE, \Phi$ *conform to Definition 2.1;*
- $\forall v_i \in V \bullet v_i \subseteq N \wedge r \in v_i$.

■

A view can be defined for any profile or, more generally, for any concern that requires only partial knowledge of the FM. As a general policy, we consider that the root is part of each view. V is a multiset to account for duplicated sets of features.

5.3.2 View specification

There are essentially two ways of specifying views, and answer *RQ5.1*. The most obvious is to enumerate, for each view, the features that appear in it—or equivalently, to tag each feature of the FM with the names of the views it belongs to. This is an *extensional definitions*. For large FMs, this might be very time-consuming and error-prone without proper tool support. A natural alternative is thus to provide a language for *intensional definitions* of views that takes advantage of the FM's tree structure to avoid lengthy enumerations. To avoid reinventing the wheel, we identified a simple subset of XPath (see Table 5.1) to fit the purpose.[3] We have chosen XPath because it was designed to navigate in tree-structures, and it is a W3C standard that has been used for more than a decade. An application to our motivating example is presented in Section 7.5. The downside of intensional definitions is that textual languages like XPath might be less affordable than graphical approaches for casual users. The consistency between the FM and the XPath expression is also harder to maintain when the model evolves. Unlike tags attached to features, XPath expressions rely on the structure of the FM and feature labels.

Table 5.1 – View query language

Path expression	Meaning
`*`	Select all the children of the current node (wildcard).
`nodename`	Select all the children with name `nodename` of the current node.
`/nodename`	Select the root node if it matches the name.
`nodename1/nodename2`	Select all the children with name `nodename2` of node `nodename1`.
`//nodename`	Select all the elements with name `nodename`, no matter where they appear.
`nodename1//nodename2`	Select all the descendants with name `nodename2` of node `nodename1`.
`path_expr1 | path_expr2`	Select all the nodes matching `path_expr1` and `path_expr2`.

In practice though, extensional and intensional definitions can be used together. XPath expressions can be generated based on interactions with a well

[3] For a formal definition, see `http://www.w3.org/TR/xpath`

designed view definition GUI rather than written by hand. Conversely, XPath expressions could be used to generate feature tags and link them to the features in the XPath expression. These links can then be used to trace changes to the FM back in the XPath expression. This way we can avoid the drawbacks of both extensional and intensional definitions.

For the formal developments that follow, we are only interested in the features contained in each view. Therefore, we will abstract from the approach chosen to specify views.

5.3.3 View coverage

An important property to be guaranteed by an FBC system is that all configuration questions be eventually answered, i.e., that a decision be made for each feature of the FM. This is the issue raised by *RQ5.2*.

In a multi-view context, it is tempting to enforce the following condition.

Definition 5.2 **Sufficient coverage condition**

For a view v of a multi-view FM u the sufficient coverage condition is:

$$\bigcup_{v \in V} v = N$$

■

Intuitively, this means that all the features appear in at least one view, hence no feature can be left undecided.[4] In our motivating example (Figure 5.2), each feature is part of a view. Hence, this condition holds: the collaborative configuration of the FM through the views will always lead to complete and valid products. This is indeed a *sufficient condition*, but not necessary since some decisions can usually be deduced from others. For instance, in the web administrator's view, if the feature *Display macro* is selected, its ancestor *Tasks* will be too, although the latter does not belong to the view.

A *necessary condition* can be defined using the notion of propositional defineability [LM08].

Definition 5.3 ***pdefines***(M, f) **(adapted from [LM08])**

For a given set N of features of u and Γ_N the Boolean encoding of u on N, a feature $f \in N$ is propositionally defined by the features in $M \subseteq N$ iff:

$$\Gamma_N \wedge \Gamma'_N[M' \leftarrow M] \vDash f \equiv f'$$

where:

[4] Note that the complete view coverage is usually assumed by mutli-view approaches (e.g. [MCMdO08]).

- Γ'_N *denotes a renaming of all the features in* Γ_N*;*
- $[M' \leftarrow M]$ *replaces all the renamed variables of* M' *by their original name in* M*.*

■

We need to ensure that the decisions on the features that do not appear in any view can be inferred from (i.e. are *propositionally defined by*) the decisions made on the features that are part of the view.

Definition 5.4 **Necessary coverage condition**

For a view v *of a multi-view FM* u *the necessary coverage condition is:*

$$\forall f \notin \bigcup_{v \in V} v \bullet \quad pdefines(\bigcup_{v \in V} v\ , f)$$

■

To evaluate *pdefines*, it suffices to translate the FM into an equivalent propositional formula (which is done in linear time [SHTB06]) and apply the SAT-based algorithm described in [LM08] (see also Section 7.7.1). This check is NP complete in theory, but this is not expected to be a problem in practice. Indeed, SAT solvers can handle FMs with thousands of features [MWC09]. To date, the largest FMs available count over 6000 features [BSL+10], which is still within the comfort zone of SAT solvers. Therefore, we are pretty confident that no performance issue would jeopardize the verification of the necessary condition, even in very large projects.

Features in $N \setminus \bigcup_{v \in V} v$ that do not satisfy the above condition will have to be integrated in existing views, or extra constraints will have to be added to determine their value.

Since the view coverage in PloneMeeting is complete, the necessary condition is trivially satisfied. However, in other domains such has operating systems, features used mostly for calculations (e.g. which boot entry should be used) are hidden to users [BSL+10]. These features cannot be part of any view. In that case, the verification of the necessary condition determines whether the value of the hidden features can be derived from the features in the views.

5.3.4 Visualisation

Views are abstract entities. To be effectively used during FBC, they need to be made concrete, i.e. visual. Henceforth, a visual representation of a view will be called a *visualisation*. The goal of a visualisation is to strike a balance between (1) showing only features that belong to a view, and (2) including features that are *not* in the view but that provide context and thereby allow

the user to make informed decisions. For instance, the *Display macro* feature is in the view of the web administrator, but its parent feature *Tasks* is not.

To tackle this problem formulated in *RQ5.1-3*, we observed the practice of the PloneMeeting developers and discussed alternative visualisations. We also inspected the approaches suggested in [ZZM08, MCMdO08]. We finally looked into filtering mechanisms provided by tools. Tools like pure::variants [psG06] or kernel configurators for operating systems (e.g. `xconfig` for Linux or `configtool` for eCos [eCo11]) provide simple filtering or search mechanisms that are similar to views on an FM. A filter is defined as a regular expression on the FM. Any feature matching the regular expression is displayed—without any control on the location of the feature in the hierarchy. Interestingly, all these approaches produce purely graphical modifications (e.g. by greying out irrelevant features) whereas cardinalities are not recomputed.

The outcome of our investigation is a set of three complementary visualisations offering different levels of details. They were built to present information on a *need-to-know* basis. They allow to regulate the amount of information displayed, and provide enhanced control over access rights. For instance, a standardised configuration menu will always display the position of the feature in the hierarchy and hide unavailable options while critical systems will conceal all the features outside a view to protect fabrication secrets. Thereby, visualisations not only propose convenient representations of a view, but also dictate what information is accessible to the stakeholder. These visualisations are depicted in Figure 5.3. The FM on the left is the same as in Figure 5.2. The darker area defines a specific view of it, called v.

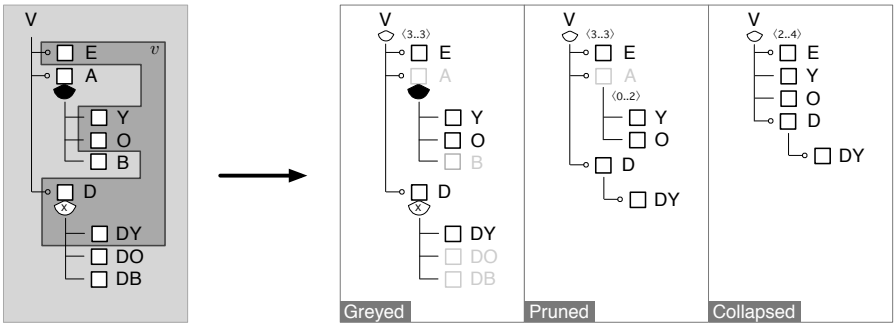


Figure 5.3 – Three alternative visualisations of FM views: greyed, pruned and collapsed.

- The *greyed* visualisation is a mere copy of the whole FM except that the features that do not belong to the view are greyed out (e.g., A, B, DO and DB). Greyed features are only displayed but cannot be manually selected/deselected.

- In the *pruned* visualisation, features that are not in the view are pruned (e.g., B, DO and DB) unless they appear on a path between a feature in the view and the root, in which case they are greyed out (e.g., A)[5].

- In the *collapsed* visualisation, all the features that do not belong to the view are pruned. A feature in the view whose parent or ancestors are pruned is connected to the closest ancestor that is still in the view. If no ancestor is in the view, the feature is directly connected to the root (e.g., Y and O).

Generating such visualisations from an FM and a view is a form of FM transformation. To implement these transformations and prove their correctness we need to formalize them.

Definition 5.5 View visualisation

The visualisation of a view v is the transformation of the original FM into a new FM such that $d_v^t = (N_v^t, r, \lambda_v^t, DE_v^t, \Phi)$, where t, the type of visualisation, can take one of three values: g (greyed), p (pruned), and c (collapsed). ■

The simplest case is the one of the greyed visualisation, since there is no transformation beyond the greying of each feature $f \notin v$ (i.e. $d_v^g = d$) . The transformations for the pruned and collapsed visualisations are respectively specified in Transformations 5.1 and 5.2. Basically, they filter nodes, remove dangling decomposition edges and adapt the cardinalities accordingly. We leave crosscutting constraints untouched in the following definitions because they are usually not displayed in FBC systems. They are reintroduced in Section 9.4 when we discuss how our toolset handles both transformations and crosscutting constraints to maintain decision consistency.

Pruned visualisation

N_v^p, the set of features in this visualisation, is the subset of N limited to features that are in v or have a descendant in v. The definition uses DE^+, the transitive closure of DE. Based on N_v^p, we remove all dangling edges, i.e. those not in $N_v^p \times N_v^p$ to create DE_v^p.

Transformation 5.1 Pruned visualisation

The transformations applied to the FM to generate the pruned visualisation are:

$$\begin{aligned} N_v^p &= \{n \in N | n \in v \vee \exists f \in v \bullet (n, f) \in DE^+\} \\ DE_v^p &= \{DE \cap (N_v^p \times N_v^p)\} \\ \lambda_v^p(f) &= (mincard_v^p(f), maxcard_v^p(f)) \end{aligned}$$

■

[5] Abstractly, when an optional feature is pruned, so is its parent non-primitive feature.

To compute the new cardinalities $\lambda_v^p(f)$, $mincard_v^p(f)$ and $maxcard_v^p(f)$ are defined as follows:

$$mincard_v^p(f) = max(0, \lambda(f).min - |orphans_v^p(f)|)$$
$$maxcard_v^p(f) = min(\lambda(f).max, |children(f)| - |orphans_v^p(f)|)$$

where $orphans_v^p(f) = children(f) \setminus N_v^p$ i.e., the set of children of f that are not in N_v^p. $\lambda(f).min$ and $\lambda(f).max$ represent the minimum and maximum values of the original cardinality, respectively. For the minimum, the difference between the cardinality and the number of orphans can be negative in some cases[6], hence the necessity to take the maximum between this value and 0. The maximum value is the maximum cardinality of f in d if the number of children in v is greater. If not, the maximum cardinality is set to the number of children that are in v.

Collapsed visualisation

The set of features N_v^c of this visualisation is simply the set of features in v. The consequence on DE_v^c is that some features have to be connected to their closest ancestor if their parent is not part of the view.

Transformation 5.2 **Collapsed visualisation**

The transformations applied to the FM to generate the collapsed visualisation are:

$$\begin{aligned} N_v^c &= v \\ DE_v^c &= \{(f,g) | f,g \in v \wedge (f,g) \in DE^+ \wedge \\ &\qquad \nexists f' \in v \bullet ((f,f') \in DE^+ \wedge (f',g) \in DE^+)\} \\ \lambda_v^c(f) &= (mincard_v^c(f), maxcard_v^c(f)) \end{aligned}$$

■

The computation of cardinalities $\lambda_v^c(f)$ is slightly more complicated than in the pruned case. Formally, $mincard_v^c(f)$ and $maxcard_v^c(f)$ are defined as follows:

$$mincard_v^c(f) = \sum min_{\lambda(f).min}(ms\_min_v^c(f))$$
$$maxcard_v^c(f) = \sum max_{\lambda(f).max}(ms\_max_v^c(f))$$

where

$$\begin{aligned} &ms\_min_v^c(f) = \\ &\quad \{mincard_v^c(g) | g \in orphans_v^c(f)\} \uplus \{1 | g \in children(f) \setminus orphans_v^c(f)\} \\ &ms\_max_v^c(f) = \\ &\quad \{maxcard_v^c(g) | g \in orphans_v^c(f)\} \uplus \{1 | g \in children(f) \setminus orphans_v^c(f)\} \end{aligned}$$

The multisets $ms\_min_v^c(f)$ and $ms\_max_v^c(f)$ collect the cardinalities of the descendants of f. The left part of the union[7] recursively collects the cardinalities of the collapsed descendants whereas the right side adds 1 for each

[6] See Section A.1 for an example.

[7] $\uplus$ is the union on multisets.

child that is in the view. The $\lambda(f).min$ minimum values of the multiset are then summed to obtain the minimum cardinality of f. The maximum value is computed similarly.

The next section illustrates how the basic steps of the transformations are applied to the motivating example presented in Section 5.2. A detailed and step-by-step explanation of the transformations performed in Figure 5.3 is reported in Section A.1.

5.4 Working Example Revisited

With the chief developer of PloneMeeting, we identified and specified three stakeholder-specific views: the coloured areas in Figure 5.2. The complete FM from which this sample is extracted is freely available online.[8] The orange area consists of the features that should be made accessible to the web administrator. Those are technical features that require a deep understanding of the inner workings of PloneMeeting. The blue area contains the features that should be made accessible to the PloneMeeting manager. They define "business" (vs. technical) configuration choices that do not evolve much at runtime and should not be edited by regular users. The red area gathers the features that should be made accessible to the end users. Their main purpose is to let users customize their web-based working environment.

These views, visually depicted as the coloured areas, can be specified with the three XPath expressions presented in Figure 5.4. The web administrator view is specified by the expression in Figure 5.4(a) and has to be interpreted as follows: the feature *Workflow and security* is in (line 1) as well as all its descendants (line 2), *Email notification* (line 3) and *Display macro* (line 4). Figure 5.4(b) and Figure 5.4(c) specify the two other views and should be understood similarly.

In Section 5.3.4, we stressed the importance of proposing alternative visualisations during FBC. We now illustrate how these transformations are applied to the PloneMeeting case. Let us focus on the transformations needed to obtain the pruned and collapsed visualisations for the web administrator. [9] The abbreviations we use for feature names are indicated in the respective figures. In the pruned case (Figure 5.5(a)), one can observe that neither the features G, D, U, V nor their descendants are in the view (see Figure 5.2). This means that they should not be accessible to the web administrator, i.e. decisions cannot be made about them (select/deselect). They are thus simply removed (pruned) from the FM. In contrast, T is not in the view but one of its children, *Display macro*, is. In that case, T is greyed out, i.e. displayed but not accessible to the

[8] `http://www.info.fundp.ac.be/~acs/tvl`

[9] The transformations of the PloneMeeting manager and user are respectively presented in Sections A.1.1 and A.1.2.

```
1       Meeting_Config/Workflow_and_security
2   |   //Workflow_and_security//*
3   |   Meeting_Config/Email_notification
4   |   Meeting_Config/Tasks/Display_macro
```

(a) XPath expression of the Web administrator view.

```
1   |   //Data/Item_insertion_algorithm
2   |   //Item_insertion_algorithm//*
3   |   Meeting_Config/User_interface
4   |   //User_interface//*
```

(b) XPath expression of the User view.

```
1       Meeting_Config/General
2   |   //General//*
3   |   Meeting_Config/Data
4   |   //Data/Meeting_attributes
5   |   //Meeting_attributes//*
6   |   //Data/Use_groups_as_categories
7   |   Meeting_Config/Workflow_and_security
8   |   //Workflow_and_security/Meeting_workflow
9   |   //Meeting_workflow//*
10  |   Meeting_Config/Email_notification
11  |   Meeting_Config/Tasks
12  |   //Tasks/Task_creator
13  |   //Task_creator//*
14  |   Meeting_Config/Votes
15  |   //Votes//*
```

(c) XPath expression of the PloneMeeting manager view.

Figure 5.4 – XPath expressions of the different views in Figure 5.2.

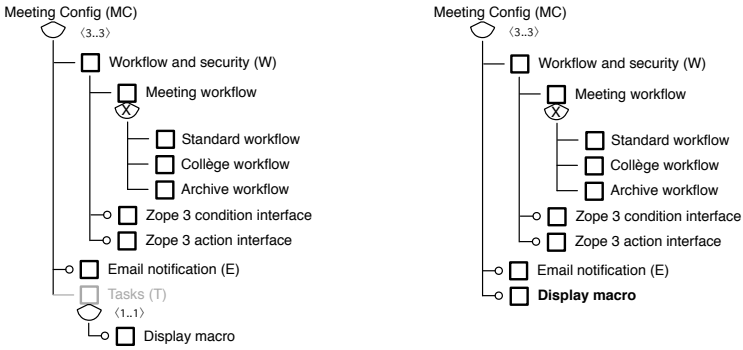
web administrator. The new set of features N_v^p thus only contains the features in Figure 5.5(a). The same holds for the decomposition edges of DE_v^p. The new cardinalities of the pruned version of the FM are calculated as follows:

$$\lambda_{WA}^p(MC) = \langle max(0, 7-4)..min(7, 7-4)\rangle = \langle 3..3\rangle$$
$$\lambda_{WA}^p(T) = \langle max(0, 2-1)..min(2, 2-1)\rangle = \langle 1..1\rangle$$

where MC has four orphans (G, D, U and V) and T only one (*Task creator*).

Obtaining the collapsed visualisation (Figure 5.5(b)) is more complex because collapsed features entail the recursive computation of cardinalities. In this particular example, the *Display macro* feature (boldfaced in Figure 5.5(b)) is the only example of a collapsed feature. Unlike in the pruned case, its parent feature T is removed from the visualisation. This means that *Display macro* is disconnected from the FM. It thus has to be linked to its closest ancestor, here MC, to keep the view consistent.

New cardinalities must be calculated to match the transformed structure of the FM. Starting from the root feature, we separate the orphans of MC (G,



(a) Pruned visualisation (b) Collapsed visualisation

Figure 5.5 – Pruned and collapsed visualisation of the web administrator.

D, U, T and V), which requires recursive calls, from its children that are in the view (W and E), which gives:

$$\begin{aligned} ms\_min^c_{WA}(MC) &= \{mincard^c_{WA}(G), mincard^c_{WA}(D), mincard^c_{WA}(U), \\ &\quad mincard^c_{WA}(T), mincard^c_{WA}(V)\} \uplus \{1,1\} \\ ms\_max^c_{WA}(MC) &= \{maxcard^c_{WA}(G), maxcard^c_{WA}(D), maxcard^c_{WA}(U), \\ &\quad maxcard^c_{WA}(T), maxcard^c_{WA}(V)\} \uplus \{1,1\} \end{aligned}$$

Only the left-hand side of the union implies a recursive call. The value for G, D, U and V is trivially 0 since neither they nor their children are in the view. T, however, has one child in the view. The cardinality of T is simple to compute since its children have no descendants, which yields:

$$\begin{aligned} mincard^c_{WA}(T) &= \textstyle\sum min_2\{0\} \uplus \{1\} = 0+1 = 1 \\ maxcard^c_{WA}(T) &= \textstyle\sum max_2\{0\} \uplus \{1\} = 0+1 = 1 \\ \lambda^c_{WA}(T) &= \langle 1..1 \rangle \end{aligned}$$

The right-hand side simply contains as many 1s as there are children of MC in the view, two in this case (W and E). The cardinality of MC in the collapsed visualisation thus gives:

$$\begin{aligned} mincard^c_{WA}(MC) &= \textstyle\sum min_7\{0,0,0,1,0\} \uplus \{1,1\} \\ &= 0+0+0+0+1+1+1 = 3 \\ maxcard^c_{WA}(MC) &= \textstyle\sum max_7\{0,0,0,1,0\} \uplus \{1,1\} \\ &= 0+0+0+0+1+1+1 = 3 \\ \lambda^c_{WA}(MC) &= \langle 3..3 \rangle \end{aligned}$$

As appears in Table 5.2, the pruned and collapsed visualisations of the sample FM of Figure 5.2 (counting 57 features), respectively the complete FM

(counting 193 features), offer significant reductions in the number of features to be handled by end-users. Regarding view definition, XPath allows relatively concise definitions (last column of Table 5.2). The number of lines needed to specify the three views of the sample and complete FMs are respectively 24 and 36. This means that for a difference of 136 features between the sample and complete FMs, only 12 additional XPath lines are needed.

Table 5.2 – Number of features for the three views and the corresponding number of XPath lines for the sample and complete FMs (S=Sample ; C=Complete).

Profile	**Greyed**		**Pruned**		**Collapsed**		**XPath**	
	S	C	S	C	S	C	S	C
Web administrator	57	193	11	48	10	47	4	5
User	57	193	20	75	19	74	5	9
PloneMeeting manager	57	193	36	120	36	120	15	22

5.5 Correctness of Transformations

It is important to demonstrate that the above transformations are correct. As mentioned earlier, FBC systems are meant to check the validity of the configuration choices based on the original global FM, not on the visualisations. Still, a proof of correctness ensures that no misleading FM constraints are shown to the stakeholders. Intuitively, the correctness criterion should state that the produced visualisations preserve a form of semantic equivalence with the original FM. We define it as follows: $[\![(N_v^t, r, \lambda_v^t, DE_v^t, \{\})]\!]_{FM} = [\![(N, r, \lambda, DE, \{\})]\!]_{FM}|_{N_v^t}$. Intuitively, the criterion means that the valid configurations one could infer from a visualisation are actually the valid configurations of the FM, when looking only at the view-specific features (hence the projection $|_{N_v^t}$), and regardless of the crosscutting constraints (hence the $\{\}$ in the two tuples). For simplicity, we ignore P which has no impact on the demonstration of the correctness.

We present below the proof of correctness for the pruned (Theorem 5.1) and collapsed (Theorem 5.2) visualisations. There is no need to prove the greyed visualisation since $d_v^g = d$.

5.5.1 Pruned transformation

Before proving the correctness in the pruned visualisation, we prove that DE_v^p in d_v^p is a prefix of DE in d, which is demonstrated in Lemma 5.1.

Definition 5.6 **Tree prefix**

Let T_1 and T_2 be trees. T_1 is a prefix *of T_2 iff there is an injection $f : N_1 \to N_2$ such that $(x, y) \in DE_1 \Leftrightarrow (f(x), f(y)) \in DE_2$ and $r_1 = f(r_2)$.* ■

Lemma 5.1 d_v^p **is a prefix of** d

The tree defined by d_v^p *is a prefix of the tree defined in* d*.*

Proof. By definition of Transformation 5.1, N_v^p contains all the features in v that appear on a path between a feature in v and the root (transitive closure of DE). Also, DE_v^p only contains decomposition edges from DE that relate features in N_v^p. Thereby, for all $(x, y) \in DE_v^p$, we have $(x, y) \in DE$, where the injection is the identity function. Furthermore, the root is also included by definition of N_v^c. □

We also need the notion of *local consistency* of an FD to account for the absence of crosscutting constraints, i.e., $\Phi = \emptyset$. Local consistency defines the satisfiability of d only in terms of the constraints imposed by decompositions, i.e. λ, and ignores crosscutting constraints.

Definition 5.7 **Local consistency**

A given $d \in \mathcal{L}_{FM}$ *such that* $\Phi = \emptyset$ *is* locally consistent *iff*

$$\forall n \in N \bullet |children(n)| \geq \lambda(n).min \wedge \lambda(n).min \leq \lambda(n).max$$

■

Also, we know from [SHTB06] that $d \in \mathcal{L}_{FM}$ is *satisfiable* if and only if $[\![d]\!]_{FM} \neq \emptyset$. From this result, we derive a corollary:

Corollary 5.1 **Local consistency satisfiability**

If $d \in \mathcal{L}_{FM}$ *is not* locally consistent*, then it is not satisfiable:* $[\![d]\!]_{FM} = \emptyset$. ■

If d is not locally consistent, then it has no valid configuration and the semantic equivalence is trivially satisfied. We demonstrate the correctness of d_v^p under the assumption that it is locally consistent.

Theorem 5.1 **Correctness of** d_v^p

If d *is locally consistent, the pruned visualisation* d_v^p *preserves the semantic equivalence with the original FM* d*:*

$$[\![(N_v^p, r, \lambda_v^p, DE_v^p, \{\})]\!]_{FM} = [\![(N, r, \lambda, DE, \{\})]\!]_{FM}|_{N_v^p}$$

Proof. We prove this theorem in two steps.

$\subseteq$ First, we prove that:

$$[\![(N_v^p, r, \lambda_v^p, DE_v^p, \{\})]\!]_{FM} \subseteq [\![(N, r, \lambda, DE, \{\})]\!]_{FM}|_{N_v^p}$$

Let us consider $c \in [\![(N_v^p, r, \lambda_v^p, DE_v^p, \{\})]\!]_{FM}$. We claim that there exists $c' \in [\![(N, r, \lambda, DE, \{\})]\!]_{FM}|_{N_v^p}$ such that $c \subseteq c'_{|N_v^p} \subseteq c'$ by local consistency. Indeed, for each $m \notin N_v^p$ and $m \in children(n)$ with $n \in N_v^p$

we have $m \in orphans_v^p(n)$, i.e. $m \notin c$. By definition we know that $mincard_v^p(n) \geq |(children(n) \cap c)| \geq maxcard_v^p(n)$. Furthermore, DE_v^p being a prefix of DE, each feature justified in c, will also be justified in c' and they both have the same root (by Lemma 5.1).

$\supseteq$ Then, we prove by *reductio ad absurdum* that:

$$[\![(N_v^p, r, \lambda_v^p, DE_v^p, \{\})]\!]_{FM} \supseteq [\![(N, r, \lambda, DE, \{\})]\!]_{FM}|_{N_v^p}$$

To do so, we will try to build a configuration c such that:

$$c \in [\![(N, r, \lambda, DE, \{\})]\!]_{FM}|_{N_v^p} \wedge c \notin [\![(N_v^p, r, \lambda_v^p, DE_v^p, \{\})]\!]_{FM}$$

To prove that such a configuration c does not exist, we test the different conditions that could lead to incompatible configurations.

1. *Different root features.* Since both d_v^p and d have the same root feature by definition of v, we know that all the configurations will have the same root feature.
2. *Every product satisfies the extra constraints.* In this case, the set of constraints is empty, hence does not influence the equality.
3. *Different decomposition edges.* We know from Lemma 5.1 that DE_v^c is a prefix of DE. Thereby, all features in N_v^p are subject to the same constraints in d_v^p and d.
4. *Different decomposition types.* We have to prove that d_v^p does not exclude configurations of d that only contain features in N_v^p. Valid configurations can be excluded if there is a feature $f \in N_v^p$ for which the interval between the minimum and maximum cardinality is reduced too much or relaxed too much.

   Let us first prove that less features than expected cannot be selected for any feature f. We know that $mincard_v^p(f) = \lambda(f).min - |orphans_v^p(f)|$ if the result is positive, which means that the recomputed value only depends on the features in N_v^p. If the result is negative, then $mincard_v^p(f) = 0$, which means that no feature in N_v^p might be selected. The cardinality is thus only reduced by the number of features outside N_v^p. Thereby, less features than required in N_v^p cannot be selected.

   More features than necessary cannot be selected either. We know that if $|children(f)| - |orphans_v^p(f)| < \lambda(f).max$ then $maxcard_v^p(f) = |children(f)| - |orphans_v^p(f)|$ which means that we can select as many features as available in N_v^p because the original cardinality is greater than the number of available children of f in N_v^p. If it is

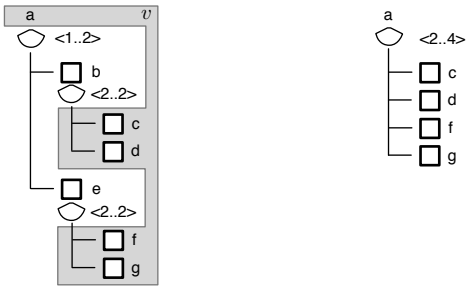
not the case, we simply have $maxcard_v^p(f) = \lambda(f).max$, which is the same condition as in d. It is thus not possible to select more features than required among those in N_v^p.

The reduction of the minimum and maximum values only depend on the number of orphans. This means that the interval cannot be altered so that it excludes configurations containing features in N_v^p.

□

5.5.2 Collapsed transformation

Unlike the pruned visualisation, the semantic equivalence in the collapsed visualisation cannot be demonstrated. Take the simple counter-example shown in Figure 5.6(a), and the collapsed visualisation of view v depicted in Figure 5.6(b). A valid configuration of the collapsed visualisation would be $\{a, d, f\}$. However, that configuration is not valid in the FM since $\{c, d\}$ and $\{f, g\}$ must always appear together in a configuration.



(a) Original FM (b) Collapsed view of the FM

Figure 5.6 – Counter-example of correctness of the collapsed visualisation.

This shows that the transformation that produces the collapsed visualisation does not preserve the semantics of the FM. Yet, we can still prove that it does not restrict the original semantics, and provides the most precise semantics expressible on v.

Definition 5.8 **Most precise semantics of a collapsed visualisation**

The most precise semantics of a collapsed visualisation of a view v on an FM d, $[\![(N_v^c, r, \lambda_v^c, DE_v^c, \{\})]\!]_{FM}$, defines the greatest possible lower bound and the smallest possible upper bound. This means that there does not exist a cardinality transformation λ^t such that:

- $\exists n \in N_v^c \bullet \lambda_v^c(n).min < \lambda^t(n).min \vee \lambda_v^c(n).max > \lambda^t(n).max$;

- $[\![(N, r, \lambda, DE, \{\})]\!]_{FM}|_{N_v^c} \subseteq [\![(N_v^c, r, \lambda^t, DE_v^c, \{\})]\!]_{FM}$, *the semantics of d is not restricted.*

■

Theorem 5.2 **Correctness of d_v^c**

If d is locally consistent (see Definition 5.7), the collapsed view d_v^p (1) does not restrict the original semantics of d, and (2) gives the most precise semantics expressible on v.

Proof. Let us demonstrate that in two steps.

(1) d_v^c does not restrict the original semantics of d. By *reductio ad absurdum* we can prove that:

$$[\![(N, r, \lambda, DE, \{\})]\!]_{FM}|_{N_v^c} \subseteq [\![(N_v^c, r, \lambda_v^c, DE_v^c, \{\})]\!]_{FM}$$

i.e. it is not possible to build a configuration c such that:

$$c \in [\![(N, r, \lambda, DE, \{\})]\!]_{FM}|_{N_v^c} \wedge c \notin [\![(N_v^c, r, \lambda_v^c, DE_v^c, \{\})]\!]_{FM}$$

1. *Every product contains the root feature.* Since both d_v^c and d have the same root feature by definition of v, we know both products have the same root feature.
2. *Every product satisfies the extra constraints.* In this case, the set of constraints is empty, hence does not influence the equality.
3. *Every feature is justified.* By definition of DE, every feature must be justified. This means that all the ancestors of a selected feature have to be selected (and nothing can be inferred about the descendants). Likewise, all the descendants of a deselected feature have to be deselected (and nothing can be inferred about the ancestors). The selection of ancestors and deselection of descendants is thus preserved in the transitive closure. Therefore, any configuration respecting DE is also valid in DE_v^c, modulo the projection on N_v^c.
4. *Every feature satisfies the decomposition type.* If c is not satisfiable in $[\![(N_v^c, r, \lambda_v^c, DE_v^c, \{\})]\!]_{FM}$, there exists $n \in N_v^c$ such that the interval of $\lambda_v^c(n)$ is too narrow. If $orphans_v^c(n) = \emptyset$ then we know by definition of Transformation 5.2 that $\lambda_v^c(n) = \lambda(n)$. The intervals are thus equivalent if f has no orphans in N_v^c.

   If $orphans_v^c(n) \neq \emptyset$, then let us consider $m \in orphans_v^c(n)$. The absence of m from N_v^c means that its children are collapsed in DE_v^c, thereby implying the recalculation of the cardinality of n. To be valid, cardinalities has to preserve the constraints imposed by the

cardinality of both n and m. The cardinality of the children of n is respected as every non-orphan is counted once and then summed up to respect the value of $\lambda(n).min$ and $\lambda(n).max$ respectively. The lower and upper bounds are also augmented with the value of the bounds of the cardinality of m. By propagating upward the cardinality constraint of m to n, one ensures that both valid combinations of children of n and m in c can be obtained. The recursion ensures that cardinalities of descendants of orphans are propagated upward. Thereby, the recomputed interval is large enough to allow all the possible configurations.

It is thus not possible to find a configuration c that is not a valid configuration of the view.

(2) d_v^c **gives the most precise semantics expressible on** v**.** From Definition 5.8, we prove by *reductio ad absurdum* that there does not exist a cardinality transformation λ^t that gives a more precise semantics than λ_v^c for any feature $n \in N_v^c$. Let us start with the minimum cardinality. If we had $\lambda_v^c(n).min < \lambda^t(n).min$, it would mean that either some orphans are missed or that less features than required are selected. The former case is not possible because $ms\_min_v^c$ takes into account all the features in N_v^c. The latter case is not possible because $mincard_v^c$ sums up the exact number of minimum cardinality of d. Likewise, for the maximum cardinality, if we had $\lambda_v^c(n).max > \lambda^t(n).max$, it would mean that either we incorrectly include orphan or we select more features than required, which cannot be.

Since we have already proven that d_v^c does not restrict the semantics of d, we know that d_v^c provides the most precise semantics.

□

Two interesting conclusions can be drawn from this latter theorem: (1) any valid configuration of the FM is a valid configuration of the collapsed view; (2) the cardinalities of the collapsed visualisation produce an under-constrained FM. This is an inevitable consequence of collapsing several descendants under the same feature. In fact, the first conclusion comes at the price of the second.

5.6 Chapter Summary

In this chapter, we have formalised an integrated solution for multi-view FBC, one of the main techniques to select product requirements during software product line engineering. Specifically, the three problems we addressed are the *specification* of a view, the *coverage* of a set of views, and the *visualisation* of a view.

View specification. Existing tools usually offer basic filtering mechanisms that rely on simple keyword-based searches, which only enable approximative navigation in the feature hierarchy. As for research papers, they do not present any concrete means to build views. To solve this issue, we have proposed alternative solutions (definition by intension vs. by extension), and developed a tool using XPath to navigate in the FM and select features.

View coverage. Most approaches overlook the notion of coverage. Those which take it into account, assume that coverage must be complete (e.g. [MBC08, MCMdO08]). The study of the coverage problem lead us to formally define *sufficient* and *necessary* coverage conditions. Both checks have been implemented in our tool.

View visualisation. Different authors have suggested different approaches to visualize views. In [ZZM08, MCMdO08], the visualisation used by the authors is comparable to the collapsed visualisation. Tools usually provide simple filtering or search mechanisms that resemble the greyed (e.g. `xconfig` and eCos) or pruned (e.g. pure::variants) visualisations. However, in both cases the result ignores FM decomposition operators and cardinalities. To address this problem, we have (1) formally defined three visualisations based on the observation made during an actual open source development project, and (2) demonstrated the correctness of these visualisations.

All these formal definitions and checks are implemented in our toolset. Chapter 9 provides details on view specification with XPath expressions, the implementation of the necessary and sufficient checks, and the maintenance of the consistency in user decisions with the three visualisations.

Chapter

6 Basic Configuration Scheduling

6.1 Multi-level Staged Configuration

According to the semantics in Section 2.3.2, an FM basically describes which configurations are allowed in the SPL, regardless of the *configuration process* to be followed for reaching one or the other configuration. Still, such a process is an integral part of SPL application engineering. Deesltra *et al.* [DS05], for instance, report that the configuration process is a "*time-consuming and expensive activity*".

Czarnecki *et al.* acknowledge the need for explicit process support, arguing that in contexts such as "*software supply chains, optimisation and policy standards*", the configuration is carried out in *stages* [CHE05]. According to the same authors, a stage can be defined "*in terms of different dimensions: phases of the product lifecycle, roles played by participants or target subsystems*". In an effort to make this explicit, they propose the concept of *multi-level staged configuration* (MLSC).

The principle of staged configuration is to remove part of the variability at each stage until only one configuration, the final product, remains. In [CHE05], the refinement itself is achieved by applying a series of syntactic transformations to the FM. Some of these transformations, such as setting the value of an attribute, involve constructs that are not formalised as part of the semantics defined in Section 2.3.2. The remaining transformations are shown in Figure 6.1. Note that they are expressed so that they conform to our semantics.

Multi-level staged configuration is the application of this idea to a series of related FMs $d_1, .., d_\ell$. Each level has its own FM, and, depending on how they are linked, the configuration of one level will induce an automatic specialisation

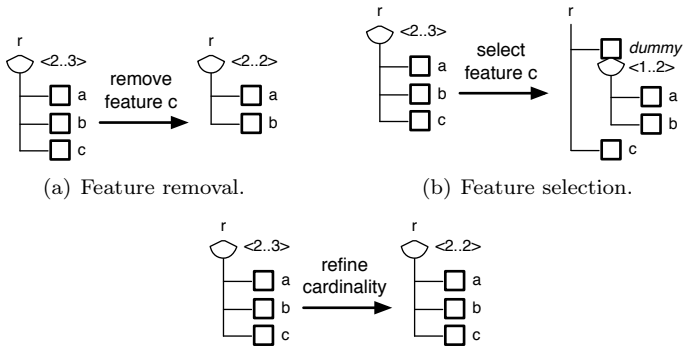


Figure 6.1 – Specialisation steps, adapted from [CHE05].

of the next level's FM. The links between models are defined explicitly through *specialisation annotations*. A specialisation annotation of a feature $f \in d_i$, ($f \in N_i$), consists of a Boolean formula ϕ over the features of $d_{i-1}(\phi \in \mathbb{B}(N_{i-1}))$. Once level $i-1$ is configured, ϕ can be evaluated on the obtained configuration $c \in [\![d_{i-1}]\!]_{FM}$, using the Boolean encoding of Section 2.4.2, i.e. a feature variable n in ϕ is *true* if and only if $n \in c$. Depending on its value and the specialisation type, the feature f will either be removed or selected through one of the first two syntactic transformations of Figure 6.1. An overview of this is shown in Table 6.1.

Table 6.1 – Possible inter-level links; original definition [CHE05](left), translation to FM semantics (right).

Spec. type	**Condition value**	**Spec. operation**	**Equivalent Boolean constraint** *with* $f \in N_i, \phi \in \mathbb{B}(N_{i-1}), c \in [\![d_{i-1}]\!]_{FM}$	
positive	*true*	select	$\phi(c) \Rightarrow f$	*Select* f, i.e. Φ_i becomes
positive	*false*	none		$\Phi_i \cup \{f\}$, if $\phi(c)$ is true.
negative	*false*	remove	$\neg\phi(c) \Rightarrow \neg f$	*Remove* f, i.e. Φ_i becomes
negative	*true*	none		$\Phi_i \cup \{\neg f\}$, if $\phi(c)$ is false.
complete	*true*	select	$\phi(c) \Leftrightarrow f$	*Select or remove* f depending on
complete	*false*	remove		the value of $\phi(c)$.

Let us illustrate this on the tax gateway component of an e-Commerce system originally introduced in [CHE05]. The component performs the calculation of taxes on orders made with the system. The customer who is going to buy such a system has the choice of three tax gateways, each offering a distinct functionality. Imagine now that there are two times at which the customer needs to decide about the gateways. The first time (Level 1) is when she purchases

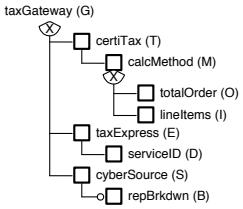


Figure 6.2 – FM example, adapted from [CHE05].

the system. All she decides at this point is which gateways will be available for use; the model that needs to be configured is the one shown on the top-left corner of Figure 6.3. Then, when the system is being deployed (Level 2), she will have to settle for one of the gateways and provide additional configuration parameters, captured by the first model in the lower-left corner of Figure 6.3. Given the inter-level links, the model in level two is automatically specialised based on the choices made in level one.

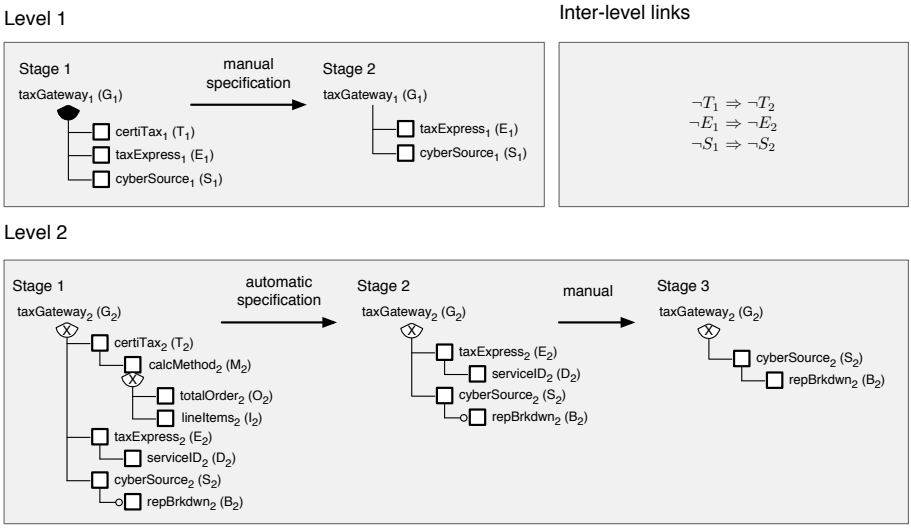


Figure 6.3 – Example of MLSC, adapted from [CHE05].

Note that even though both models in the example are very similar, they need not be so. Also note that the original paper mentions the possibility, that several configuration levels might run in parallel. It applies, for instance,

if levels represent independent decisions that need to be taken by different people. As we show later on, such situations give rise to interesting decision problems.

The MLSC approach, as it appears in [CHE05], is entirely based on *syntactic* transformations. This makes it difficult to decide things such as whether two levels A and B are commutative (executing A before B leaves the same variability as executing B before A). This is the main motivation for defining a formal semantics, as follows in the next section. To respect the initial definition of MLSC, this chapter uses the term level. In the next chapter, the level is replaced by the more general concept of view. For now, it suffices to see a level as a special kind of view on the FM.

6.2 Dynamic FM Semantics ($[\![.]\!]_{\mathbf{MLSC}}$)

We introduce the dynamic FM semantics in two steps. The first, Section 6.2.1, defines the basic staged configuration semantics; the second, Section 6.2.2, adds the multi-level aspect.

6.2.1 Staged configuration semantics

Since we first want to model the different stages of the configuration process, regardless of levels, the syntactic domain $\mathcal{L}_{FM}$ will remain as defined in Section 2.3.2. The semantic domain, however, changes since we want to capture the idea of building a product by deciding incrementally which configuration to retain and which to exclude.

Indeed, we consider the semantic domain to be the set of all possible *configuration paths* that can be taken when building a configuration. Along each such path, the initially full *configuration space* ($[\![d]\!]_{FM}$) progressively shrinks (i.e., configurations are discarded) until only one configuration is left, at which point the path stops. Note that in this work, we thus assume that we are dealing with *finite* configuration processes where, once a unique configuration is reached, it remains the same for the rest of the life of the application. Definitions 6.1 and 6.3 formalise the intuition we just gave.

Definition 6.1 **Dynamic semantic domain $\mathcal{S}_{CP}$**

Given a finite set of features N, a configuration path *π is a finite sequence $\pi = \sigma_1 ... \sigma_n$ of length $n > 0$, where each $\sigma_i \in \mathcal{P}(\mathcal{P}(N))$ is called a* stage*. If we call the set of such paths C, then $\mathcal{S}_{CP} = \mathcal{P}(C)$.* ■

The following definition will be convenient when expressing properties of configuration paths.

Definition 6.2 **Path notation and helpers**

- ϵ *denotes the empty sequence*
- $last(\sigma_1 ... \sigma_k) = \sigma_k$

■

Definition 6.3 **Staged configuration semantics** $[\![d]\!]_{CP}$

Given an FM $d \in \mathcal{L}_{FM}$, $[\![d]\!]_{CP}$ *returns all* legal *paths* π *(noted* $\pi \in [\![d]\!]_{CP}$*, or* $\pi \models_{CP} d$*) such that*

(6.3.1) $\sigma_1 = [\![d]\!]_{FM}$

(6.3.2) $\forall i \in \{2..n\} \bullet \sigma_i \subset \sigma_{i-1}$

(6.3.3) $|\sigma_n| = 1$

■

Note that this semantics is not meant to be used as an implementation directly, for it would be very inefficient. This is usual for denotational semantics which are essentially meant to serve as a conceptual foundation and a reference for checking the conformance of tools [Sto77]. Along these lines, we draw the reader's attention to condition (6.3.2) which will force compliant configuration tools to let users make only "useful" configuration choices, that is, choices that effectively eliminate configurations. At the same time, tools must ensure that a legal product eventually remains reachable given the choices made, as requested by condition (6.3.3).

As an illustration, Figure 6.4 shows an example FM and its legal paths. A number of properties can be derived from the above definitions.

Theorem 6.1 **Properties of configuration paths**

(6.1.1) $[\![d]\!]_{FM} = \emptyset \Leftrightarrow [\![d]\!]_{CP} = \emptyset$

(6.1.2) $\forall c \in [\![d]\!]_{FM} \bullet \exists \pi \in [\![d]\!]_{CP} \bullet last(\pi) = \{c\}$

(6.1.3) $\forall \pi \in [\![d]\!]_{CP} \bullet \exists c \in [\![d]\!]_{FM} \bullet last(\pi) = \{c\}$

Contrary to what intuition might suggest, (6.1.2) and (6.1.3) do not imply that $|[\![d]\!]_{FM}| = |[\![d]\!]_{CP}|$, they merely say that every configuration allowed by the FM can be reached as part of a configuration path, and that each configuration path ends with a configuration allowed by the FM.

Czarnecki *et al.* [CHE05] define a number of transformation rules that are to be used when specialising an FM, three of which are shown in Figure 6.1. With the formal semantics, we can now verify whether these rules are expressively complete, i.e. whether is it always possible to express a σ_i $(i > 1)$ through the application of the three transformation rules.

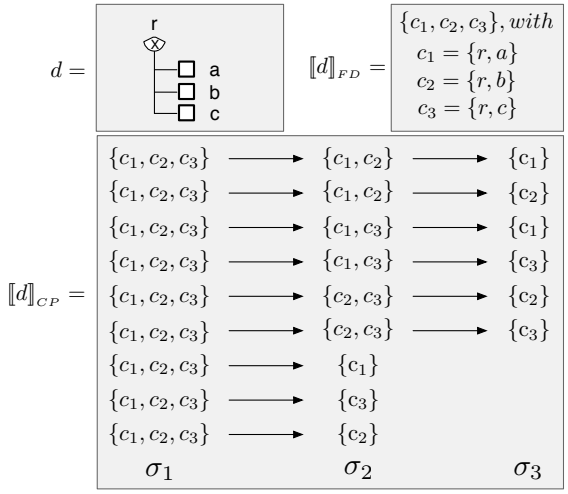


Figure 6.4 – The staged configuration semantics illustrated.

Theorem 6.2 **Incompleteness of transformation rules**

The transformation rules shown in Figure 6.1 are expressively incomplete wrt. the semantics of Definition 6.3.

Proof. Consider a model consisting of a parent feature $\langle 2..2 \rangle$-decomposed with three children a, b, c. It is not possible to express the σ_i consisting of $\{a, b\}$ and $\{b, c\}$, by starting at $\sigma_1 = \big\{\{a, b\}, \{a, c\}, \{b, c\}\big\}$ and using the proposed transformation rules (since removing one feature will always result in removing at least two configurations). □

Note that this is not necessarily a bad thing, since Czarnecki *et al.* probably chose to only include transformation steps that implement the most frequent usages. However, the practical consequences of this limitation need to be assessed empirically.

6.2.2 Adding levels

Section 6.2.1 only deals with dynamic aspects of staged configuration of a single model. If we want to generalise this to MLSC, we need to consider multiple models and links between them. To do so, there are two possibilities: (1) define a new abstract syntax, that makes the set of diagrams and the links between them explicit, or (2) encode this information using the syntax we already have.

We chose the latter option, mainly because it allows to reuse most of the existing definitions and infrastructure, and because it can more easily be generalised. Indeed, a set of FMs, linked with conditions of the types defined in Table 6.1, can be represented as a single big FM. The root of each individual FM becomes a child of the root of the combined FM. The root is *and*-decomposed and the inter-level links are represented by Boolean formulae. To keep track of where the features in the combined FM came from, the level information will be made explicit as follows. Figure 6.5 illustrates how the example of Figure 6.3 is represented in $\mathcal{L}_{\text{MLSC}}$.

Definition 6.4 **Dynamic syntactic domain $\mathcal{L}_{\text{MLSC}}$**

$\mathcal{L}_{MLSC}$ consists of 7-tuples $(N, P, L, r, \lambda, DE, \Phi)$, where:

- *$N, P, r, \lambda, DE, \Phi$ follow Definition 2.1;*
- *$L = L_1...L_\ell$ is a partition of $N \setminus \{r\}$ representing the list of levels.*

So that each $d \in \mathcal{L}_{MLSC}$ satisfies the well-formedness rules of Definition 2.1, has an and-decomposed root, and each level $L_i \in L$:

- *is connected through exactly one node to the global root: $\exists! n \in L_i \bullet (r, n) \in DE$, noted hereafter $root(L_i)$;*
- *does not share decomposition edges with other levels (except for the root): $\forall (n, n') \in DE \bullet (n \in L_i \Leftrightarrow n' \in L_i) \vee (n = r \wedge n' = root(L_i))$;*
- *is itself a valid FM, i.e. $(L_i,\ P \cap L_i,\ root(L_i),\ \lambda \cap (L_i \to \mathbb{N} \times \mathbb{N}),\ DE \cap (L_i \times L_i),\ \emptyset)$ satisfies Definition 2.1.*[1]

■

Given the new syntactic domain, we need to revise the semantic function. As for the semantic domain, it can remain the same, since we still want to reason about the possible configuration paths of an FM. The addition of multiple levels, however, requires us to reconsider what a *legal* configuration path is. Indeed, we want to restrict the configuration paths to those that obey the levels specified in the FM. Formally, this is defined as follows.

Definition 6.5 **Dynamic FM semantics $[\![d]\!]_{\text{MLSC}}$**

Given an FM $d \in \mathcal{L}_{MLSC}$, $[\![d]\!]_{MLSC}$ returns all paths π that are legal wrt. Definition 6.3, i.e. $\pi \in [\![d]\!]_{CP}$, and for which there exists a legal level arrangement*, that is π, except for its initial stage, can be divided into ℓ $(= |L|)$* levels*: $\pi = \sigma_1 \Sigma_1 .. \Sigma_\ell$, each Σ_i corresponding to an L_i such that:*

(6.5.1) $|final(\Sigma_i)_{|L_i}| = 1$, *i.e., Σ_i is fully configured;*

[1] The set of constraints here is empty because it is not needed for validity verification.

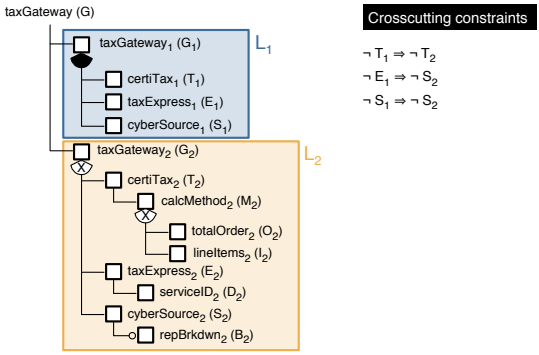


Figure 6.5 – Example of Figure 6.3 in $\mathcal{L}_{\text{MLSC}}$.

(6.5.2) $\forall \sigma_j \sigma_{j+1} \bullet \pi = ...\sigma_j\sigma_{j+1}... \wedge \sigma_{j+1} \in \Sigma_i$, *we have*

$$(\sigma_j \setminus \sigma_{j+1})_{|L_i} \subseteq (\sigma_{j\,|L_i} \setminus \sigma_{j+1\,|L_i}).$$

As before, this will be noted $\pi \in [\![d]\!]_{MLSC}$, *or* $\pi \models_{MLSC} d$. ■

We made use of the following helper.

Definition 6.6 **Final stage of a level** Σ_i

For $i = 1..\ell$,

$$final(\Sigma_i) \triangleq \begin{cases} last(\Sigma_i) & \text{if } \Sigma_i \neq \epsilon \\ final(\Sigma_{i-1}) & \text{if } \Sigma_i = \epsilon \text{ and } i > 1 \\ \sigma_1 & \text{if } \Sigma_i = \epsilon \text{ and } i = 1 \end{cases}$$

■

The rule (6.5.2) expresses the fact that each configuration deleted from σ_j (i.e. $c \in \sigma_j \setminus \sigma_{j+1}$) during level L_i must be necessary to delete one of the configurations of L_i that are deleted during this stage. In other words, the set of *deleted* configurations needs to be included in the set of *deletable* configurations for that level. The deletable configurations in a stage of a level are those that indeed remove configurations pertaining to that level (hence: first reduce to the level, then subtract), whereas the deleted configurations in a stage of a level are all those that were removed (hence: first subtract, then reduce to level to make comparable). Intuitively, this corresponds to the fact that each decision has to affect only the level at which it is taken.

6.3 Illustration of $[\![.]\!]_{\text{MLSC}}$

Let us illustrate this with the FM of Figure 6.5, which we will call d, itself being based on the example of Figure 6.3 in Section 6.1. The semantic domain of $[\![d]\!]_{\text{MLSC}}$ still consists of configuration paths, i.e. it did not change from those of $[\![d]\!]_{CP}$ shown in Figure 6.4. Yet, given that $[\![d]\!]_{\text{MLSC}}$ takes into account the levels defined for d, not all possible configuration paths given by $[\![d]\!]_{CP}$ are legal. Namely, those that do not conform to rules (6.5.1) and (6.5.2) need to be discarded. This is depicted in Figure 6.6, where the upper box denotes the staged configuration semantics of d ($[\![d]\!]_{CP}$), and the lower box denotes $[\![d]\!]_{\text{MLSC}}$, i.e. the subset of $[\![d]\!]_{CP}$ that conforms to Definition 6.5.

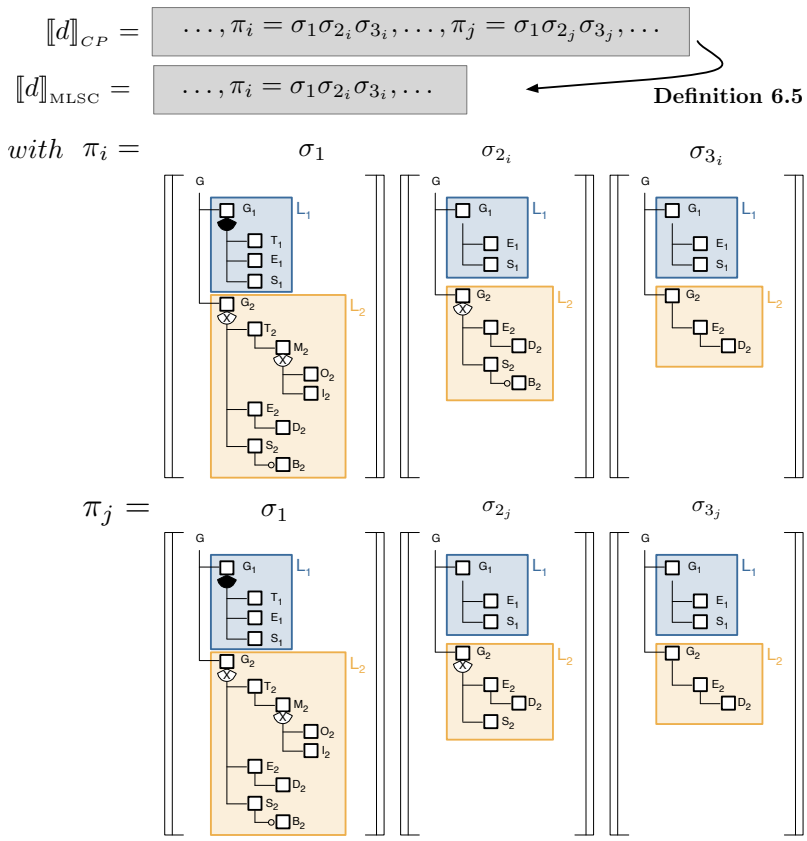
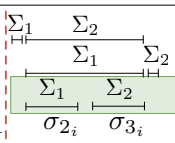
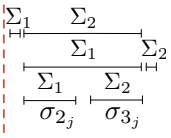


Figure 6.6 – Example of Figure 6.3 in $[\![d]\!]_{CP}$ and $[\![d]\!]_{\text{MLSC}}$.

We now zoom in on two configuration paths $\pi_i, \pi_j \in [\![d]\!]_{CP}$, shown with

Table 6.2 – Validation of level arrangements.

Level arrangement for path	rule 6.5.1	rule 6.5.2
$\pi_i = \sigma_1$ 	false	/
	true	false
	true	true
$\pi_j = \sigma_1$ 	false	/
	true	false
	true	false

the help of intermediate FMs in the lower part of Figure 6.6. As noted in Figure 6.6, π_j is not part of $[\![d]\!]_{\text{MLSC}}$ since it violates Definition 6.5, whereas π_i satisfies it and is kept. The rationale for this is provided in Table 6.2. Indeed, for π_j, there exists no level arrangement that would satisfy both rules (6.5.1) and (6.5.2). This is because in σ_{2_j}, it is not allowed to remove the feature B_2, since it belongs to L_2, and L_1 is not yet completed. Therefore, either there is still some variability left in the FM at the end of the level, which is thus not fully configured (the first possible arrangement of π_j in Table 6.2 violates rule (6.5.1)), or the set of deleted configurations is greater than the set of deletable configurations (the other two arrangements of π_j in Table 6.2, which violate rule (6.5.2)). For π_i, on the other hand, a valid level arrangement exists and is indicated by the highlighted line in Table 6.2. All details for this illustration are provided in Appendix B.1.

6.4 Properties of the Semantics

In Definition 6.5, we require that it has to be possible to divide a configuration path into level arrangements that satisfy certain properties. The definition being purely declarative, it does not allow an immediate conclusion as to how many valid level arrangements one might find. The following two theorems show that there is exactly one.

Theorem 6.3 **Properties of level arrangements**

Given a diagram $d \in \mathcal{L}_{MLSC}$, each configuration path $\pi \in [\![d]\!]_{MLSC}$ with $\Sigma_1..\Sigma_\ell$ as a valid level arrangement satisfies the following properties:

(6.3.a) *If $\sigma_j \in \Sigma_i$ then $\forall k < j \bullet |\sigma_{k\,|L_i}| > |\sigma_{j\,|L_i}|$.*

(6.3.b) *If $\sigma_j \in \Sigma_i$ and $\sigma_j \neq last(\Sigma_i)$ then $|\sigma_{j\,|L_i}| > 1$.*

(6.3.c) *If* $|\sigma_{j\,|L_i}| = 1$ *then* $\forall k > j \bullet \sigma_k \notin \Sigma_i$.

(6.3.d) *If* $|\sigma_{j\,|L_i}| = 1$ *then* $\forall k > j \bullet |\sigma_{k\,|L_i}| = 1$.

Proof.

(6.3.a) Because of rule (6.3.2), at each stage at least one configuration is deleted: $\forall k < j \bullet |\sigma_k| > |\sigma_j|$. In addition, (6.5.2) guarantees that those deleted pertain to the level including the stage: $\forall k < j \bullet |\sigma_{k\,|L_i}| > |\sigma_{j\,|L_i}|$.

(6.3.b) Immediate consequence of the previous property combined with rule (6.5.1) saying that $|last(\Sigma_i)_{\,|L_i}| = 1$.

(6.3.c) Take the case $k = j+1$. Suppose that $\sigma_{j+1} \in \Sigma_i$. Because of property (6.3.a), this would mean that $|\sigma_{j\,|L_i}| > |\sigma_{j+1\,|L_i}|$, i.e. $|\sigma_{j+1\,|L_i}| = 0$, which is impossible since $\forall c \in \sigma_j \bullet root(L_i) \in c$. The cases $k > j+1$ are similar.

(6.3.d) $|\sigma_{j\,|L_i}| = 1$ means that all configurations $c \in \sigma_j$ contain $\sigma_{j\,|L_i}$. Given rule (6.3.2), only full configurations can be removed, hence the property.

□

Theorem 6.4 **Uniqueness of level arrangement**

For any diagram $d \in \mathcal{L}_{MLSC}$*, a level arrangement for a configuration path* $\pi \in [\![d]\!]_{MLSC}$ *is unique.*

Proof. Let us suppose that it is possible to find a diagram d that has a valid configuration path $\pi \in [\![d]\!]_{\text{MLSC}}$ with more than one level arrangement. Note, that in that case, we need more than one level to begin with: $|L| > 1$. Without loss of generality, each $\pi \in [\![d]\!]_{\text{MLSC}}$ with multiple arrangements falls into one of the following three categories.

1. π consists of a single stage $\pi = \sigma_1$: these π cannot have multiple level arrangements since every level is empty.

2. σ_2 can be assigned to two different levels

$\pi =$	σ_1	σ_2	...
Level arrangement 1		Σ_i	...
Level arrangement 2		Σ_{i+k}	...

   with $\Sigma_1..\Sigma_{i-1} = \epsilon$, $k \geq 1$ and $\Sigma_{i+1}..\Sigma_{i+k-1} = \epsilon$. This situation is impossible; if both assignments obey Definition 6.5, i.e. they both satisfy rules 6.5.1 and 6.5.2, they necessarily exclude each other:

If arrangement 1 is legal, then $|\sigma_{1\,|L_i}| > 1$ by property (6.3.a) meaning that arrangement 2 would violate rule (6.5.1).

If arrangement 2 is legal, then $|\sigma_{1\,|L_i}| = 1$ by rule (6.5.1) and arrangement 1 would violate property (6.3.c).

3. σ_j, with $j > 2$, can be assigned to two different levels

$\pi =$	$\ldots$	σ_{j-1}	σ_j	$\ldots$
Level arrangement 1	$\ldots$	$\ldots$	Σ_i	$\ldots$
Level arrangement 2	$\ldots$	$\ldots$	Σ_{i+k}	$\ldots$

with $k \geq 1$ and $\Sigma_{i+1}..\Sigma_{i+k-1} = \epsilon$. This case is similar to (b), i.e. if both arrangements are legal, they also exclude each other:

If arrangement 1 is legal, then $|\sigma_{j-1\,|L_i}| > 1$ by property (6.3.a). Level arrangement 2 is impossible since it would violate rule (6.5.1).

If arrangement 2 is legal, then $|\sigma_{j-1\,|L_i}| = 1$ by rule (6.5.1). Given property (6.3.c), arrangement 1 is then impossible.

Since these are all the situations that might occur, it is impossible for a $\pi \in [\![d]\!]_{\text{MLSC}}$ to have multiple level arrangements. □

An immediate consequence of this result is that it is possible to determine a legal arrangement *a posteriori*, i.e. given a configuration path, it is possible to determine a unique level arrangement describing the process followed for its creation. Therefore, levels need not be part of the semantic domain. This result leads to the following definition.

Definition 6.7 Subsequence of level arrangement

Given an FM d and $L_i \in L, \pi \in [\![d]\!]_{MLSC}$, $sub(L_i, \pi)$ denotes the subsequence Σ_i of π pertaining to level L_i for the level arrangement of π that satisfies Definition 6.5. ■

Continuing with Definition 6.5, remember that rule (6.5.2) requires that every *deleted* configuration be *deletable* in the stage of the associated level. An immediate consequence of this is that, unless we have reached the end of the configuration path, the set of *deletable* configurations must not be empty, established in Theorem 6.5. A second theorem, Theorem 6.6, shows that configurations that are deletable in a stage, are necessarily deleted in this stage.

Theorem 6.5 Sufficient replacement of rule (6.5.2)

A necessary, but not sufficient replacement for rule (6.5.2) is that $(\sigma_{j\,|L_i} \setminus \sigma_{j+1\,|L_i}) \neq \emptyset$.

Proof. Immediate via *reductio ad absurdum.* □

Theorem 6.6 **Equality of deletable and deleted decision sets**

For rule (6.5.2) of Definition 6.5 holds

$$\begin{aligned}(\sigma_j \setminus \sigma_{j+1})_{|L_i} \;&\subseteq\; (\sigma_{j\,|L_i} \setminus \sigma_{j+1\,|L_i}) \\ \Rightarrow (\sigma_j \setminus \sigma_{j+1})_{|L_i} \;&=\; (\sigma_{j\,|L_i} \setminus \sigma_{j+1\,|L_i}).\end{aligned}$$

Proof. In Theorem B.1 included in Appendix B.2, we prove that always

$$(\sigma_j \setminus \sigma_{j+1})_{|L_i} \;\supseteq\; (\sigma_{j\,|L_i} \setminus \sigma_{j+1\,|L_i}).$$

which means that if in addition $(\sigma_j \setminus \sigma_{j+1})_{|L_i} \subseteq (\sigma_{j\,|L_i} \setminus \sigma_{j+1\,|L_i})$ holds, both sets are equal. □

In Theorem 6.2, Section 6.2.1, we showed that the transformation rules of Figure 6.1, i.e. those proposed in [CHE05] that relate to constructs formalised in the abstract syntax of Definition 6.4, are not expressively complete wrt. the basic staged configuration semantics of Definition 6.3. The two following theorems provide analogous results, but for the dynamic FM semantics. Basically, the property still holds for the dynamic FM semantics of Definition 6.5, and a similar property holds for the proposed inter-level link types of Table 6.1.

Theorem 6.7 **Incompleteness of transformation rules**

The transformation rules shown in Figure 6.1 are expressively incomplete wrt. the semantics of Definition 6.5.

Proof. We can easily construct an example for $\mathcal{L}_{\text{MLSC}}$; it suffices to take the FM used to prove Theorem 6.2 and to consider it as the sole level of a diagram. From there on, the proof is the same. □

Theorem 6.8 **Incompleteness of inter-level link types**

The inter-level link types proposed in [CHE05] are expressively incomplete wrt. the semantics of Definition 6.5.

Proof. Basically, the proposed inter-level link types always have a sole feature on their right-hand side. It is thus impossible, for example, to express the fact that if some condition ϕ is satisfied for level L_i, all configurations of level L_{i+1} that have f will be excluded if they also have f' (i.e. $\phi \Rightarrow (f' \Rightarrow \neg f)$). □

During the elaboration of an FM, the analyst needs to decide the order of the levels. Since this order determines which configuration paths are considered legal, an important information at this point is whether two levels can be interchanged, i.e. whether they are *commutative*. Two levels are called commutative if the tails of the configuration paths, starting with the later one, are the same for either order of them.

Definition 6.8 **Commutative levels**

Given an FM d, so that $L = [..L_a..L_b..]$, L_a *and* L_b *are said to be* commutative, *iff the following holds:*

$$\begin{aligned} &\{final(sub(L_b, \pi))|\pi \in [\![d]\!]_{MLSC}\} \\ &= \quad \{final(sub(L_a, \pi))|\pi \in [\![d']\!]_{MLSC}\}, \end{aligned}$$

where $d' = d$ *except for* $L' = L$ *with* L_a *and* L_b *inverted.* ■

6.5 Chapter Summary

This chapter introduced a dynamic formal semantics for FMs that allows reasoning about its configuration paths, i.e. the configuration process, rather than only about its allowed configurations. Extending the basic dynamic semantics with levels yielded a semantics for MLSC. The contribution is therefore a precise and formal account of MLSC that makes the original definition [CHE05] more explicit and reveals some of its subtleties and incompletenesses. Based on the semantics we showed some interesting properties of configuration paths.

However, the sequential process enforced by MLSC has its limitations. In practice, the strictly sequential ordering of configuration activities rarely holds. In the next chapter, we relax that assumption and describe a combined formalism to control and reason about non-linear configuration processes.

Chapter

7

Advanced Configuration Scheduling

7.1 Relaxing Restrictions on Levels

The semantics defined in Section 6.2 inherits from the original definition of MLSC [CHE05] the assumption that levels are configured one after the other in a strict order until the final configuration is obtained. We will gradually lift these hypotheses, and discuss their implications on the semantics.

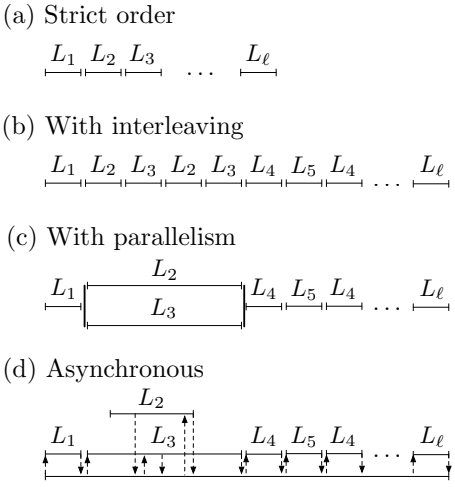


Figure 7.1 – Illustrations of possible level orderings.

Interleaved levels. As stated in Definition 6.4, for an FM $d \in \mathcal{L}_{\text{MLSC}}$, levels are specified with a strict order that must be followed during the configuration process, as illustrated in Figure 7.1(a). In a realistic project, however, this restriction might prove to be too strong. It is more likely that several stakeholders may want to take decisions pertaining to their respective levels during the same period of time, for instance because they are independent. A graphical illustration is provided in Figure 7.1(b). Nevertheless, project management might still want to avoid chaotic or arbitrary procedures, by imposing a more loose order which allows for interleaving of levels.

Because of the interleaving, this intuition cannot be captured by any kind of order on the set of levels. A loose order with interleaving is a specification of the possible level arrangements, rather than an order on the levels themselves. Definition 6.5 requires for a configuration path to be legal, that there must be an assignment of stages to levels that satisfies certain properties. Intuitively, one can think of this assignment as a labelling of the transitions between stages by levels, i.e.

$$\sigma_i \xrightarrow{L_j} \sigma_{i+1} \stackrel{\triangle}{=} \sigma_{i+1} \in \Sigma_j.$$

Now consider the sequence consisting only of the L_j's noted on top of the arrows. With the current definition, this sequence is required to be of the form

$$L_1..L_1..L_2..L_2....L_\ell.$$

Interleaving means that we want to also allow, for instance,

$$L_1..L_2..L_3..L_2..L_3..L_4..L_5..L_4....L_\ell,$$ [1]

while still being able to specify which interleavings are allowed (e.g. only L_2/L_3 and L_4/L_5), and what the minimal order is (e.g. a strict order from L_6 on). One way to do that is with a regular expression over the L_i's, such as

$$L_1^*(L_2|L_3)^*(L_4|L_5)^*L_6^*...L_\ell^*$$

that expresses the example constraints noted above, or

$$L_1^*L_2^*...L_\ell^*$$

corresponding to the constraint as it is in the current definition. Rather than specifying an order on L, Definition 6.4 would require to specify a regular expression over elements of L that has to be enforced in the semantic function of Definition 6.5. Given that regular expressions can be expressed equivalently by automata, alternative formalisms like statecharts, 1-safe Petri nets or other

[1] Where L_1 now has to be fully configured at its last occurrence in the sequence rather than when L_2 starts.

process diagrams could also be used to specify the allowed sequences. Namely a statechart, where a state corresponds to the configuration of a level and transitions denote configuration sequences, might allow stakeholders and managers with a less formal background to specify the configuration process intuitively.

Interleaving also gives rise to a number of interesting analysis properties. During the elaboration of the FM, mainly when defining the configuration process, it is important to know which levels are safe to be interleaved. To this end, it is interesting to know whether two levels are independent, i.e. whether there are direct/indirect inter-level links between them, or whether they influence the same features in a subsequent level.

Parallel levels. Actually, interleaving, with well-done tool support, already allows for pseudo-parallelism; similar to how a single core processor allows multiple programs to run in parallel by interleaving their instructions. It requires, however, the model to be accessible for configuration simultaneously from different places. Google Docs, and the recent *software-as-a-service* trend, show how this is possible even within a browser, yet these approaches generally require a live Internet connection. Lifting this barrier to distributed offline collaboration would make the semantics even more fit to real scenarios, since parallel configuration does happen in the real world, for instance in automotive software engineering [CHE05]. Figure 7.1 illustrates the differences between strict order, interleaving and parallelism graphically.

Asynchronous levels. The advantage of parallel levels is that distributed groups can work independently on their local copies. The model of parallelism introduced previously, however, still assumes a certain amount of co-ordination, namely at the fork and merge points. This can lead to problems; imagine, for instance, that L_2 and L_3 run in parallel. If the configuration of L_3 takes longer than expected, the subsequent levels will have to wait for L_3 to finish, even though L_2 is already configured.

This problem could be solved by considering a completely asynchronous approach, as shown in Figure 7.1(d). There is a central base model, but instead of executing configurations on the base model, each level is locally configured and merges back its decision into the base model on the fly. This way, L_2 can be merged back to the central model even before L_3 is finished. If merges are assumed to work in the other direction as well, then this can also reduce the potential for conflict, since each level can merge its changes back to the central model as it progresses with the configuration (note that, if a merge is done for every stage, this is roughly equal to the interleaving mode). This asynchronous level model actually corresponds to how popular SCM systems work.

Crosscutting levels. In Definition 6.5, we require that each level has to be an FM in itself, with no sharing of features or decomposition edges between levels. It could be imagined to lift this hypothesis, too, and thereby allow levels to be crosscutting. Indeed, Lee *et al.* suggested to group features into *binding units*, denoting major functionalities of the system, so that nodes

shared between binding units allow for additional constraints [LK06]. Given that binding units can be represented by levels in our semantics, it seems that such an extension would make sense. Implications on the semantics, however, have to be considered carefully.

To support these extensions, our semantics needs to be adapted in several ways. To give a concrete sense to these extensions, we first illustrate the limitations of MLSC and levels on a real-world SPL: the CFDP developed by Spacebel (Section 7.2). We then introduce the workflow language (Section 7.3) that, together with generic views on the FM, defines a new practical formalism called *feature-based configuration workflow* (Section 7.4). The application of the formalism to the CFDP case is described in Section 7.5. Next, we study the desired properties of feature-based configuration workflows (Sections 7.6 and 7.7). We conclude with an experiment (Section 7.8) and threats to validity (Section 7.9).

7.2 Working Example: CFDP

Spacebel is a Belgian software company developing software for the aerospace industry. We collaborate with Spacebel on the development of an SPL for flight-grade libraries implementing the CSSDS File Delivery Protocol (CFDP) [DPC+08, Con07]. The CFDP is a file transfer protocol specifically designed for space requirements, such as long transmission delays. The protocol was designed to cover the needs of a broad range of space missions. For a given mission, however, only part of the protocol is used, and since resources for onboard software are limited, all CFDP implementations are actually mission-specific. Spacebel thus built an SPL of CFDP libraries, where each library can be tailored to the needs of a specific mission.

The FM of the CFDP library product line counts 80 features, has a maximal depth of four, and contains ten additional constraints. A simplified excerpt of this FM appears in Figure 7.2. The principal features provide the capability to send (*Send*) and receive (*Receive*) files. The *Extended* feature allows a device to send and receive packets via other devices (such as a lander transmitting via an orbiting satellite). The *Reboot* feature allows the protocol to resume transfers safely after a sudden system reboot. PUS stands for Packet Utilisation Standard, part of the ESA standard for transport of telemetry and telecommand data (TMTC). The *PUS* feature implements the CFDP related services of this standard.

The extensions to MLSC that we propose in this chapter are motivated by the problems we encountered when applying it to automate the CFDP configuration process. A number of different stakeholders participate in the configuration of a mission-specific CFDP library. Initially, *Spacebel* decides which features are mature enough for the mission (flight-grade *vs.* ground sta-

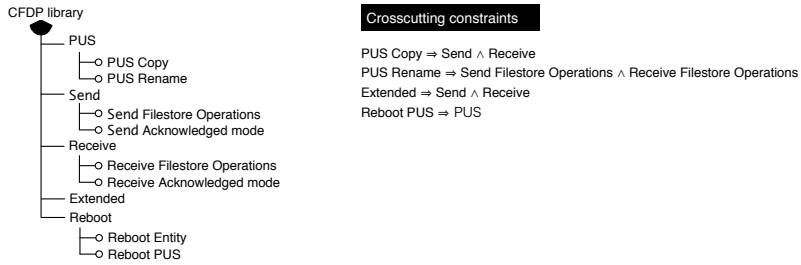


Figure 7.2 – Sample FM of the CFDP library.

tion), while leaving as much variability as possible. In certain cases, a *Reseller* negotiates the contract. The *Reseller* can, depending on the contract, or for other commercial reasons, further configure the product. The configuration task is then passed on to the company that builds the software for the mission. The *system engineer* (*SE*) makes initial high-level choices and passes the task of refining these choices onto the *network integrator* (*NWI*) and the *TMTC integrator* (*TTI*). These two configure in parallel the parts of the library they are responsible for. For technical reasons (e.g., reduction of available CPU time due to overruns by other components), integrators might have to come back to the SE to warrant or change some feature selections. The configuration can therefore be an iterative procedure until a final configuration is determined, and the library is finally delivered by Spacebel.

MLSC, as defined in Chapter 6, is too restrictive to account for a complex scenario such as this one. Indeed, the original MLSC approach assumes the process to be purely sequential, but this is not the case here: (1) the *NWI* and *TTI* perform configuration *in parallel*, (2) the configuration by the *reseller* is *optional*, (3) the FM of *Spacebel* is *not fully configured* when its intervention is over, and (4) *configuration iterates* between *SE* and *NWI/TTI*.

7.3 YAWL: A Walkthrough

As the limitations we just identified indicate, we need support for modelling and enforcing configuration processes that are more complex than mere sequences. Workflow modelling languages and tools serve this purpose.

Among the possible options, we picked YAWL as it is formal [vdAtH05], has extensive tool support [vdAADTH04], is known to cover a large variety of *workflow modelling patterns* [vdAtHKB03], can be mapped from other languages (e.g. BPMN, BPEL, and activity diagrams [WVvdA$^+$09]), and has been successfully applied to a wide variety of industrial settings. In [vdAtHKB03], van der Aalst *et al.* conduct a study of 20 workflow patterns, and compare

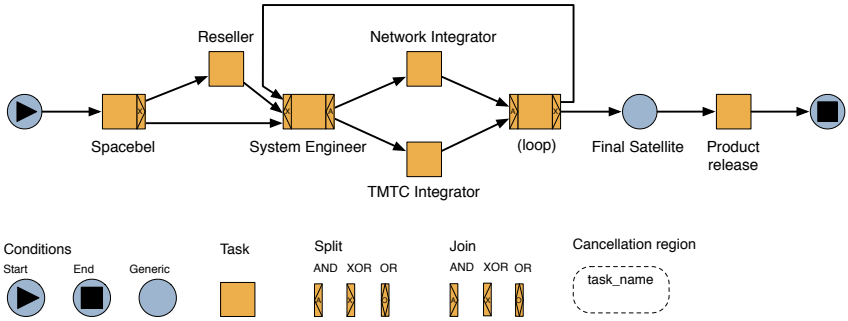


Figure 7.3 – CFDP configuration workflow.

the coverage of 15 workflow management systems and associated languages. They conclude that the suitability of those systems *"leaves much to be desired"* [vdAtHKB03], and propose YAWL as an alternative covering a large set of modelling patterns.

The scenario introduced in the previous section can be represented by the YAWL workflow shown in Figure 7.3. YAWL is inspired by Petri nets. Its principal constructs are (conditions) and [tasks], which roughly correspond to *places* and *transitions* in Petri nets. There are two special conditions, (begin) and (end), and generic conditions (e.g., (Final Satellite)). In Figure 7.3, each task, except for [(loop)] and [Product release], denotes a configuration activity, and is annotated with the name of the stakeholder performing the configuration. [Spacebel] is split in two with a *xor*-split, meaning that only one of the outgoing transitions is executed, which captures the optional nature of [Reseller]. [System engineer] joins both paths and then splits again, but this time with an *and*-split, meaning that the [Network] and [TMTC integrator] run in parallel. From there, the configuration process is either finished ([Product Release]) or continues with [System engineer].

Formally, a workflow is defined as follows.

Definition 7.1 Workflow [vdAtH05]

A workflow w is defined as a tuple $(C, b, e, F, T, split, join)$ where C denotes the set of conditions, $b \in C$ is the unique begin condition, $e \in C$ the unique end condition (single entry and exit points of the workflow), $F \subseteq (C \setminus \{e\} \times T) \cup (T \times C \setminus \{b\}) \cup (T \times T)$ is the flow relation between conditions and tasks, T denotes the set of tasks. split (respectively join) is the function determining the type of the task split (respectively join) behaviour, i.e. OR, AND, XOR.

■

The semantics of w, noted $[\![w]\!]_{YAWL}$, is a transition system $(S, \rightarrow)$, where each state $s \in S$ contains a set of tokens which marks the active *conditions* and *tasks*.

For the sake of understandability, we restrict the definition to the basic concepts of YAWL workflows. Definition 7.1 is actually a simplification of the original version of extended workflow nets [vdAtH05]. Composite tasks, multiple instances, and cancellation regions were discarded. Cancellation regions will be reintroduced in Section 7.7.1 for analyses. We will review these restrictions in Section 7.9.

7.4 Feature-based Configuration Workflow ($[\![.]\!]_{FCW}$)

Having introduced FMs and YAWL, we now introduce the new combined formalism of feature-based configuration workflows (FCWs) and illustrate it with the Spacebel case.

In a nutshell, an FCW is a workflow, such as the one shown in Figure 7.3, where a task is associated with a set of features. In MLSC, each set would be an independent level, or FM, which is connected to the other levels through inter-level links. However, that representation is not always the most suitable. In the CFDP case, for instance, every set of features is a projection on the FM in Figure 7.2. Building separate FMs and the extra-constraints would put a heavy burden on the designers. Even with proper tool support, such representation would clutter the definition of the FCW. Therefore, instead of levels, we use the more generic concept of view (see Chapter 5), that can accommodate both independent FMs—as in MLSC— and projections on a single model without introducing unnecessary duplication.

Using YAWL to model the CFDP configuration process allows us to overcome the restrictions of MLSC outlined in Sections 7.1 and 7.2. From a purely structural viewpoint, it provides an immediate solution to the representation of *parallel views* through *and*-split, *optional views* through *xor*-split and *iterative configurations* through backward transitions.

Relaxing the limitation that FM views be completely configured before passing on to the next view is of a more fundamental nature. Ideally, the formalism should be flexible enough to overcome this limitation but rigid enough to enforce the time when views have to be configured. This is achieved by specifying, separately for each view, the task in which it can be configured, and the point at which configuration has to be finished. This point is represented by a condition in the workflow.

We now provide a formal syntax and semantics for FCWs that follows that intuitive description.

Definition 7.2 **Abstract syntax $\mathcal{L}_{FCW}$**

An FCW $m \in \mathcal{L}_{FCW}$ is a tuple $(w, d, start, stop)$ such that:

- *w is a workflow, i.e.* $w = (C, b, e, F, T, join, split)$.
- *u is a multi-view FM, i.e.* $u = (N, r, \lambda, DE, \Phi, V)$.
- $start : V \to T$ *is a total injective function assigning each view to a task (its* start*) in the workflow. start being injective, it means that at most one view can be linked to a task, and each view has exactly one task.*
- $stop : V \to C$ *is a total function assigning each view to a condition (its* stop*) in the workflow.*

■

Intuitively, the *start of a view* is the only task of the workflow during which the associated view can be configured, while the *stop of a view* denotes the point at which the configuration of a view needs to be done. The reason why the start and stop of a view are dissociated, is to be able to capture cases where the partial configuration of a view is completed by subsequent views, or where the configuration iterates, such as between the *SE*, the *NWI*, and the *TTI* in the Spacebel case.

As for *MLSC*, the semantic domain of the *FCW* language is also based on the notion of *configuration path* (see Definition 6.1 and 6.3).

The semantics of an FCW is the set of legal configuration paths (see 7.3.A in Definition 7.3) which follow a valid sequence of workflow states (7.3.B). Intuitively, this means those configuration paths where the products eliminated in a step pertain to the view whose task is being executed (7.3.B.2), and where the stops encountered during the workflow execution are respected (7.3.B.3). This intuition is formalised by saying that each stage σ of the configuration path can be associated to a state s in the workflow, i.e. a sequence φ of pairs (σ, s), that verifies the two above conditions and a minor well-formedness condition (7.3.B.1).

Definition 7.3 **FCW Semantics** $[\![m]\!]_{FCW}$

For $m \in \mathcal{L}_{FCW}$, $[\![m]\!]_{FCW}$ *returns the set of paths* $\pi \in \mathcal{S}_{CP}$ *such that* $\pi = \sigma_1 \dots \sigma_n$ *for which there is a valid sequence of YAWL states* $\rho \in [\![w]\!]_{YAWL}$ *with* $\rho = s_1 \to \dots \to s_k$ *such that:*

(7.3.A) *let* u' *be* u *with* $V' = V \setminus V_{skip}$, π *is a legal configuration path of* u'*;*

(7.3.B) $\exists$ *a sequence* $\varphi : (\sigma_1, s_1), \dots, (\sigma_i, s_i)$ *such that:*

(7.3.B.1) either the configuration path or the workflow sequence evolve stepwise:

$$\forall \dots (\sigma_i, s_i)(\sigma_{i+1}, s_{i+1}) \dots \in \varphi$$
$$(\sigma_i = \sigma_{i+1} \wedge s_i \to s_{i+1} \in \rho) \vee (s_i = s_{i+1})$$

(7.3.B.2) only the active views are configured at a time:

$$\begin{array}{l}\forall \ldots (\sigma_i, s_i)(\sigma_{i+1}, s_i) \ldots \in \varphi \\ \quad \bullet\ (\sigma_i \setminus \sigma_{i+1})_{\,|active(s_i)} \neq \emptyset \\ \qquad \wedge\ \ (\sigma_i \setminus \sigma_{i+1})_{\,|active(s_i)} \subseteq \\ \qquad\quad (\sigma_{i\,|active(s_i)} \setminus \sigma_{i+1\,|active(s_i)})\end{array}$$

(7.3.B.3) all the stops of the views are satisfied:

$$\forall (\sigma_i, s_i) \in \varphi \bullet \forall v \in stops(s_i) \bullet |\sigma_{i\,|v}| = 1$$

■

where:

- $c(s)$ is the set of conditions active in state s.
- $t(s)$ is the set of tasks active in state s.
- $active(s)$ returns the union of the views active in a given state s:

$$active(s) \triangleq \{\bigcup v_i | start(v_i) \in t(s)\}$$

- $V_{skip} = V \setminus (\bigcup_{s \in \rho}\ active(s))$ is the set of views that do not appear in ρ.
- Conversely, $V_{do} = V \setminus V_{skip}$ is the set of views that do appear in ρ.
- *stops* returns the set of views that should be fully configured in a given state s:

$$stops(s) \triangleq \{v_i \in V_{do} | stop(v_i) \in c(s)\}$$

- *starts* returns the set of views that should be configured in a given state s:

$$starts(s) \triangleq \{v_i \in V_{do} | start(v_i) \in t(s)\}$$

7.5 Working Example Revisited

Section 7.2 introduced the configuration scenario of the CFDP. This scenario is now re-used to illustrate our definition of FCW. Figure 7.4 depicts three types of artefacts: (1) the views, (2) the configuration workflow, and (3) the mappings between both. Note that the concrete syntax presented here is used for illustrative purpose only and is not meant to be prescriptive.

The *view decomposition* is based on the sample diagram of Figure 7.2. The decomposition of the original FM produces five views, each accounting for the roles and responsibilities defined in Section 7.2. All the views are rendered with the *pruned* visualisation (see Section 5.3.4).

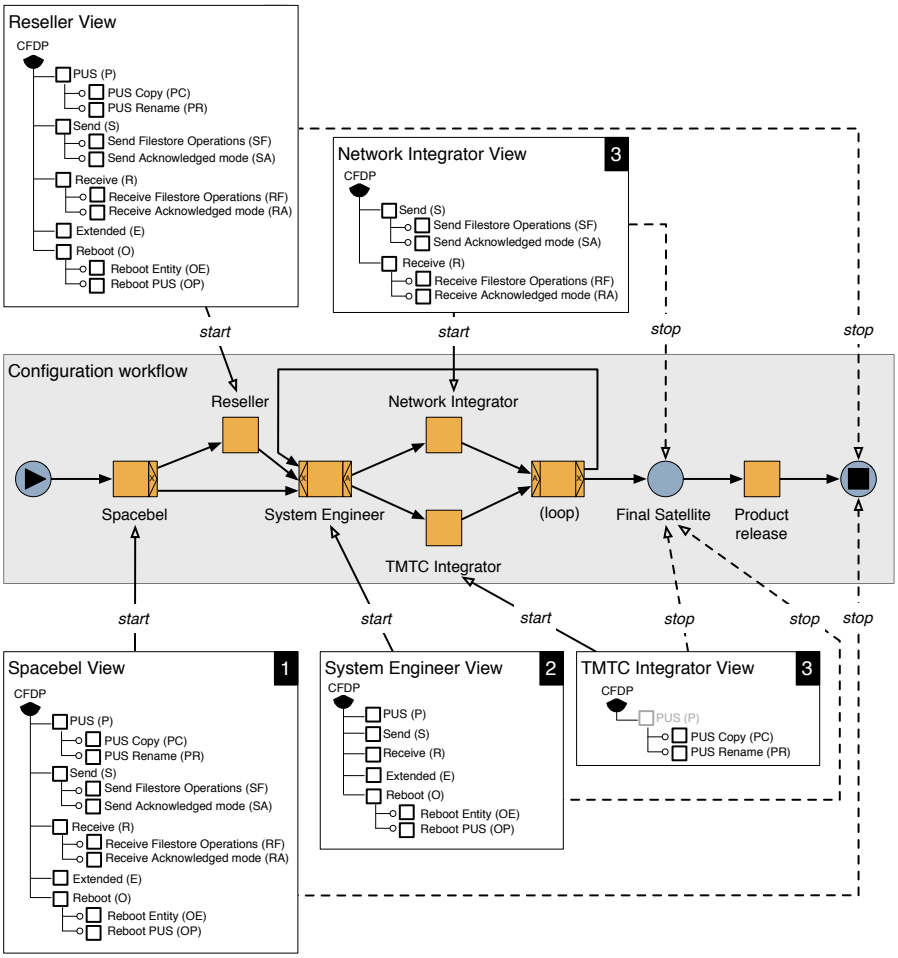


Figure 7.4 – Example of FCW applied to the Spacebel scenario.

The *configuration workflow* is the one discussed in Section 7.3.

The *mapping* of the views to the tasks follows directly from the view decomposition. The mapping to the stops, however, requires some further explanation. The (Final Satellite) stop, to which three views (*System Engineer*, *Network Integrator* and *TMTC Integrator*) point, indicates the group formed by these modules: as long as the stop is not satisfied, the loop has to continue. The two other views, including the optional *Reseller* view, map onto (end), the second stop. Note that, if the [Reseller] task is not executed, its features are not part of the configuration paths, meaning that constraint 7.3.B.3

is automatically satisfied.

Figure 7.5 illustrates how this FCW can be executed, which eventually results in a fully configured product. The workflow starts with the Spacebel task (indicated by the arrow in Figure 7.5(a)), where the responsible person decides that the *PUS Copy* feature shall not be included. As a consequence, the feature becomes unavailable in the *TMTC Integrator* view. The next task is the System Engineer , who decides to include all the features that are available to him as shown in Figure 7.5(b). This decision again causes other features to be selected automatically, including those of the previous module. Finally, Network - and TMTC Integrator finalise the configuration process in parallel. Their choices eliminate all remaining variability, meaning that both stops are satisfied, and that the workflow has reached the end.

7.6 Desired Properties

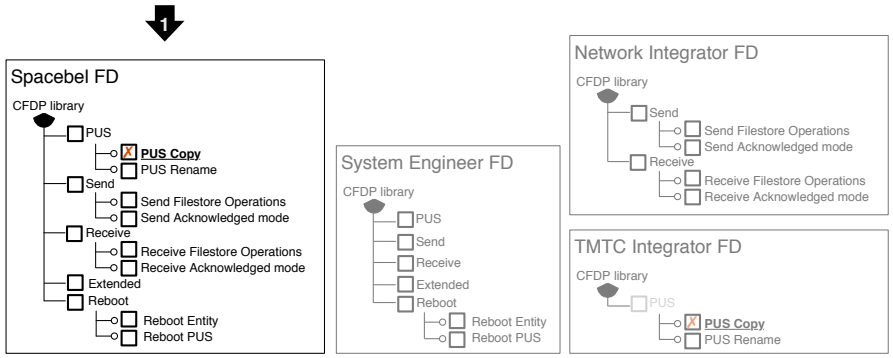
The semantics of FCWs voluntarily leaves freedom to FCW modellers. That freedom can come at the price of incomplete products or unsatisfiable FCWs. The semantics of FCWs defines the properties of a valid execution of the workflow. However, not all executions of the workflow are semantically valid. For instance, an execution could terminate with an incomplete configuration because an optional configuration task was ignored by mistake, or stall because the current decisions do not satisfy a stop condition. There is currently no way to avoid faulty executions. To prevent such problems, undesired behaviours should be diagnosed at *design* time and prohibited at *execution* time.

To avoid inconsistent executions of an FCW, we have to guarantee the *satisfiability* of an FCW, and that any decision left open during the configuration can be *postponed*. To achieve that goal, we build upon previous work on FM [BSRC10] and workflow [vdAADTH04, Wyn06, WVvdA+09, tHvdAAR09] analyses. In this section, we formally define the satisfiability and postponability properties. The next section will present algorithms implementing their verification.

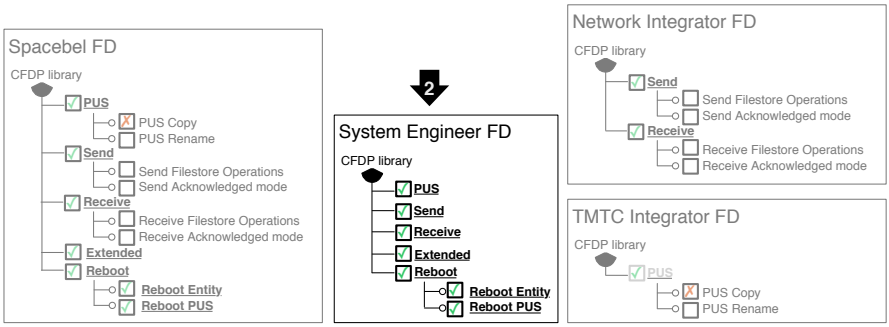
7.6.1 Satisfiability

To define the FCW satisfiability, we first build upon the definition of workflow (weak) soundness [vdAtH05]. The (weak) soundness property ensures that (1) every execution of the workflow completes, (2) no task is still running when the execution stops, and (3) every task is executable.

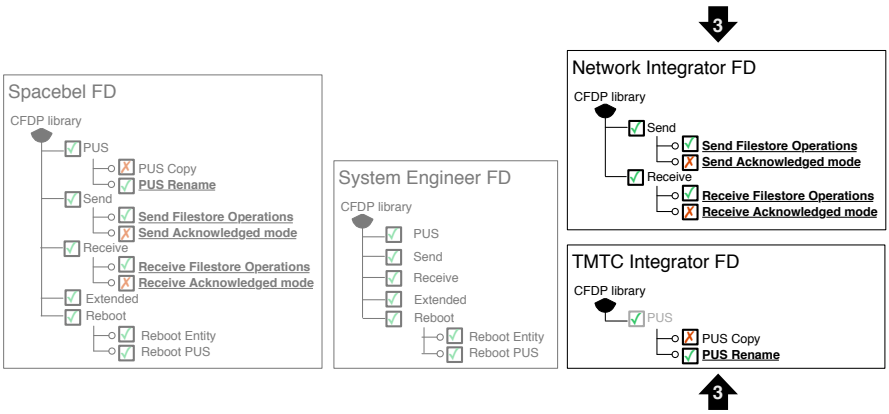
In the following definitions, $s_i \xrightarrow{X} s_j$ extends the $\rightarrow$ used in Definition 7.3. X can be $*$ (possibly empty sequence of states that lead from s_i to s_j), $+$ (non empty sequence of states that lead from s_i to s_j), or $x \in s_j \wedge x \notin s_i$, where x is either a task or a condition. We also denote s_b the first state of



(a) Step 1 – Configuration by Spacebel



(b) Step 2 – Configuration by the System Engineer



(c) Step 3 – Parallel configuration by the Network and TMTC Integrators

Figure 7.5 – Example of valid module configuration derivable from Figure 7.4.

the execution and s_e the final state of the execution such that $\{b\} = s_b$ and $\{e\} = s_e$, respectively. We finally write $p\bullet = \{q|(p,q) \in F\}$, the set of output places of p, and $\bullet q = \{p|(p,q) \in F\}$, the set of input places of q.

Definition 7.4 **Workflow (Weak) Soundness (adapted from [vdAtH05])**

A workflow w is (weakly) sound if and only if:

(7.4.A) *w has the* option to complete*:*

$$\forall s \bullet (s_b \xrightarrow{*} s) \Rightarrow (s \xrightarrow{*} s_e)$$

The option to complete *property might be undecidable if the state space is infinite. In such cases, w is said to have a weak option to complete, meaning that the process might complete in some cases. If only the weak option to complete holds, then the workflow is weakly sound if the two following properties are satisfied.*

(7.4.B) *w has* proper completion *when the last state contains only the output condition:*

$$\forall s \bullet (s_b \xrightarrow{*} s \wedge s_e \subseteq s) \Rightarrow (s = s_e)$$

(7.4.C) *w has no* no dead transition *when, for every transition, there is a sequence in which it is fired:*

$$\forall (f_x, f_y) \in F \bullet \exists s_i, s_{i+1} \bullet f_x \in s_i \wedge f_y \in s_{i+1}$$

■

The soundness property determines whether the configuration process can complete properly and every configuration task can be executed, i.e., is not dead. These are essential requirements for an FCW. However, neither soundness nor $\mathcal{L}_{FCW}$ restrict the placement of a stop wrt. its task. As explained in the previous section, this was done intentionally to account for cases in which consecutive FMs are refinements of an initial FM. The additional flexibility comes at the cost of having to detect unsafe models.

The position of the stop is critical as it defines when the associated views must be completely configured. By construction, a stop can come before, after, or during the execution of the tasks linked to those views, or simply never be active. Furthermore, the position of a stop can vary from one run to another. In Figure 7.6 for instance, all the runs of the workflow reported in Figure 7.7 are perfectly valid wrt. YAWL's semantics. The position of the stop varies from one case to the other, which yields completely different conditions on the views. If the stop is anterior or concurrent to its tasks, then all the features in the associated views have to be propositionally defined by decisions made earlier in the run, e.g., through crosscutting constraints. This is very unlikely

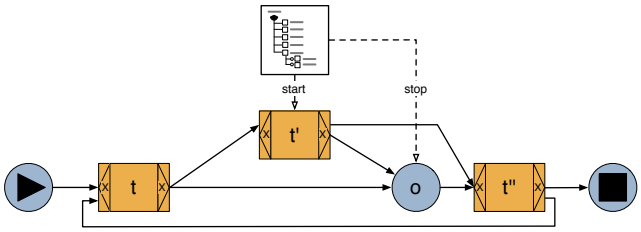


Figure 7.6 – Example of FCW with a floating stop o for task t'.

to happen in practice considering the complexity of the models. It also makes the view useless as it will never be manually configured—one thus looses the advantage of postponable decisions. Therefore, an FCW is *safe* when it is always possible to reach a complete configuration (7.5.A), and a stop always follows its tasks (7.5.B).

Definition 7.5 **FCW safety**

A given FCW $m \in \mathcal{L}_{FCW}$ *is* safe *iff :*

(7.5.A) *the views linked to the mandatory configuration tasks completely cover the FM:* $\forall \rho \bullet \forall f \notin V_{mandatory} \bullet \quad pdefines(V_{mandatory}, f)$

(7.5.B) *the stops always follow their tasks:*

$$\forall t \bullet \exists v \in V \wedge t = start(v) \bullet follows(t, stop(start^{-1}(t)))$$

■

where $V_{mandatory} = \{\bigcup v | \bigcap_{\rho} \bigcup_{s \in rho} s \bullet start(v) \in t(s)\}$.

A stop correctly *follows* its task when two conditions are satisfied. First, if a task is in a sequence, then its stop must be too (7.6.A). Secondly, the stop can never precede the first occurrence of the task (7.6.B). These two conditions are captured in the following definition.

Definition 7.6 ***follows***(t, c)

For all $\rho \in [\![w]\!]_{YAWL}$ *that contain* t*, the condition* c follows *the task* t *iff:*

(7.6.A) c *occurs after* t*:* $\exists s_i, s_j \bullet t \in t(s_i) \Rightarrow (c \in c(s_j) \wedge i \geq j)$.

(7.6.B) c *does not occur before the first occurrence of* t*:*

$$\nexists s_i, s_j, s_k, s_l \bullet s_b \xrightarrow{*\setminus t} s_i \xrightarrow{c} s_j \xrightarrow{*} s_k \xrightarrow{t} s_l$$

■

Workflow soundness and safety, together with view coverage (see Definition 5.2 and 5.4), defines the satisfiability of an FCW.

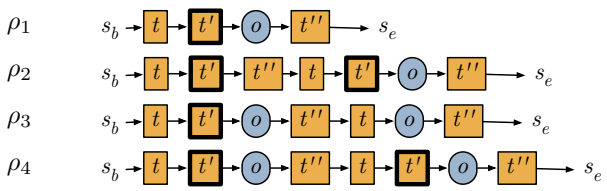
Definition 7.7 **FCW satisfiability**

An FCW $m \in \mathcal{L}_{FCW}$, is satisfiable iff:

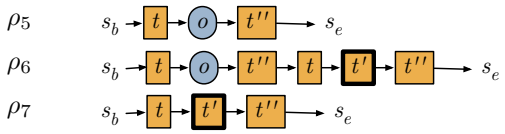
- *w is sound;*
- *the view coverage of u is complete;*
- *m is safe.*

■

To illustrate Definition 7.7, let us come back to the FCW in Figure 7.6. Figure 7.7(a) shows some examples of valid runs. One can observe that for every occurrence of t' there is a posterior occurrence of o. ρ_4 might seem peculiar as t' both precedes and follows o. In fact, the second occurrence of t' has no impact on the configuration of v since it is already fully configured—enforced by the stop condition o. It also conforms to Definition 7.5 since there is at least one occurrence of t' before o.



(a) Examples of legal runs of tasks t, t' and the stop o.



(b) Examples of illegal runs of tasks t, t' and the stop o.

Figure 7.7 – Examples of legal and illegal runs of Figure 7.6.

In contrast, the two runs in Figure 7.7(b) are illegal. ρ_5 only contains o. Definition 7.5.B is satisfied but not Definition 7.5.A; the view is mandatory but not configured. ρ_6 is obviously illegal because t' does not appear before o (Definition 7.6.B). To be legal, it should have appeared at least once before o, as in ρ_1 or ρ_2 for instance. The problem with ρ_7 is the absence of o, which does not allow to check the proper completion of t' (Definition 7.6.A).

7.6.2 Postponable decisions

A satisfiable FCW might still not complete. In Definition 7.3, condition 7.3.B.3 requires that when a stop is reached, the views linked to it have to be fully configured. Satisfiability does not enforce that condition. The reason is that the decision to include or exclude a feature can only be made at execution time. The challenge is to make sure that the view is fully configured *before* the stop condition is reached in order to prevent deadlocks [CHH09a], as in ρ_5 or ρ_6.

To avoid deadlocks, all the features of a view that are left *open* (undecided) at the end of the configuration task have to be configured before the stop is reached. To be *postponable*, all the open features have to be *propositionally defined* by features in views attached to tasks that appear on paths between the current task and its stop. For instance, in Figure 7.4, the open features in the view of the system engineer can be defined by the views of the TMTC and network integrators. The views of Spacebel and the reseller cannot be used here because they *never* follow the system engineer in the workflow.

The set of features that could propositionally define a feature f configurable in a task t is called a *dependency set*. We distinguish two types of dependency sets. The *weak dependency set* (*WD*) contains all the features attached to places reachable from t, irrespective of the split of the tasks in the workflow. It is weak because not all the features contained in it might be reached in a specific path. The *strong dependency set* (*SD*) takes into account splits so that *SD* only contains features that will always be reachable, no matter what path is taken. These two sets are defined below where $F_{p_1,p_2} \subseteq F$ denotes the set of paths between $p_1, p_2 \in C \cup T$. For simplicity, we assume that $start^{-1}(p)$ returns an empty set when undefined.

Definition 7.8 **Weak dependency set (WD_{p_1,p_2})**

For a given $p_1 \in T \cup C$, the weak dependency set *of features reachable from p_1 until p_2 is reached, called $WD_{p_1,p_2} \subseteq N$, is defined by:*

$$WD_{p_1,p_2} = \begin{cases} \bigcup_{q \in p_1\bullet} WD_{q,p_2} & \text{if} \quad p_1 \in C \\ \bigcup_{q \in p_1\bullet} WD_{q,p_2} \cup start^{-1}(q) & \text{if} \quad p_1 \in T \end{cases}$$

subject to $(p,q) \in F_{p_1,p_2}$, i.e., we only keep paths that lead from p_1 to p_2. ■

Definition 7.9 **Strong dependency set (SD_{p_1,p_2})**

For a given $p_1 \in T \cup C$, the strong dependency set *of features reachable from p_1 until p_2 is reached, called $SD_{p_1,p_2} \subseteq N$, is defined by:*

$$SD_{p_1,p_2} = \begin{cases} \bigcap_{q \in p_1\bullet} SD_{q,p_2} & \text{if} \quad p_1 \in C \\ \bigcup_{q \in p_1\bullet} SD_{q,p_2} \cup start^{-1}(q) & \text{if} \quad p_1 \in T \wedge split(t) = AND \\ \bigcap_{q \in p_1\bullet} SD_{q,p_2} \cup start^{-1}(q) & otherwise \end{cases}$$

subject to $(p,q) \in F_{p_1,p_2}$. ■

Both WD_{p_1,p_2} and SD_{p_1,p_2} filter out the features that do not belong to paths between the starting place p_1 and the end place p_2. It is obvious that these sets make sense only if there is a path between p_1 and p_2. This is not a problem here because we focus only on paths between a task t and the associated stop o, which we know exist if the FCW is satisfiable.

Based on that definition, we can define the concepts of *weakly* and *strongly* open feature. Intuitively, a strongly open feature will always be decided upon before the stop condition is reached whereas a weakly open view might not.

Definition 7.10 **Weakly open feature**

Given a view $v \in V$, its start $t = start(v)$ and stop $o = stop(v)$, a feature $f \in v$ is weakly open *iff it is undecided and pdefines$(WD_{t,o}, f)$.* ■

Definition 7.11 **Strongly open feature**

Given a view $v \in V$, its start $t = start(v)$ and stop $o = stop(v)$, a feature $f \in v$ is strongly open *iff it is undecided and pdefines$(SD_{t,o}, f)$.* ■

By extension, a view is weakly (respectively strongly) open when at least one of its features is weakly (respectively strongly) open.

In the Spacebel example, $WD_{TMTC\ Integrator,Final\ Satellite}$ contains the features from the views of the system engineer, the network integrator, and the TMTC integrator itself, while $SD_{TMTC\ Integrator,Final\ Satellite}$ is empty. In the former case, it means that the open features of the TMTC integrator could be defined through some iterations in the loop. In the latter case, the *xor*-split of (loop) will return an empty set because the intersection with the set of features from (Final satellite) is empty. The result is that no feature can be left open in the view of the TMTC integrator if one wants to be sure that the stop condition is satisfied. That information is particularly useful in practice because it can either reveal a design flaw or be used as a warning. In that particular case, it might indicate that users must perform another iteration if they want open features to be decided upon, and the stop condition to be satisfied.

7.7 Analysis of Feature-based Configuration Workflows

Equipped with these formal specifications, we are set to build the algorithms that automate the analysis of FCWs. As for the properties, we first explain how the satisfiability of an FCW can be verified, and then how open features are checked at execution time.

7.7.1 Satisfiability

In Definition 7.7, we have seen that an FCW is satisfiable when its workflow is sound, the view coverage is complete, and it is safe. Soundness analysis has already been fully implemented in the YAWL tool suite [tHvdAAR09], and is not further explored here. Similarly, both the necessary and sufficient view coverage conditions have already been implemented (see Chapters 5 and 9). We simply recall here that $\mathit{pdefines}(M, f)$ can be obtained with a single SAT check:

$$\neg\text{SAT}(\Gamma_N \wedge \Gamma'_N[M' \leftarrow M] \wedge ((f \wedge \neg f') \vee (\neg f \vee f')))$$

If the formula is satisfiable, it is possible to find two different configurations of d (Γ_N and Γ'_N), such that the assignment of the features in M are the same but f and f' are different. In other words, it means that for the same assignment of M, the values of f and f' can differ, hence M does not propositionally define f. Since SAT returns `true` if M does *not* propositionally define f, the returned value is negated to match the definition of $\mathit{pdefines}(M, f)$.

To verify the safety property we have to check that (1) the mandatory tasks can completely configure the FM, (2) a stop does not precede its tasks, and (3) the stop of a task always appears in the run. Interestingly, the verification of the safety property can be reduced to soundness verification. The idea is to transform the original workflow such that the soundness analysis of the transformed workflow reveals whether some designated tasks prevent the option to complete (they cause a deadlock) or are dead. The type of error determines which property is violated. Besides capitalizing on optimisations of the reasoning engine, that approach does not require adaptation to the tool, and can be applied to any workflow language supporting *xor*-split/joins, *and*-split/joins, and cancellation regions.

A transformed workflow is generated for every view in the FCW. First, the addition of elements in the workflow is linear in the number of views. However, the complexity of the soundness analysis increases exponentially because of the extra loops and cancellation regions added by the transformation. Generating separate workflows limits the growth of the state space. Secondly, we want to reduce the sensitivity of the soundness analysis to workflow constructs. As we will see in Section 7.8, loops and *or*-joins have a detrimental effect on performance and correctness. The individual analysis guarantees that views not affected by these constructs are correctly analysed. The division of the analysis thus increases scalability, precision, and reliability.

The transformation of a workflow is a seven-step process. Each transformation is informally discussed here and illustrated on the example in Figure 7.6, transformed in Figure 7.8. The transformation is formally defined in Algorithm 2 in Appendix C.

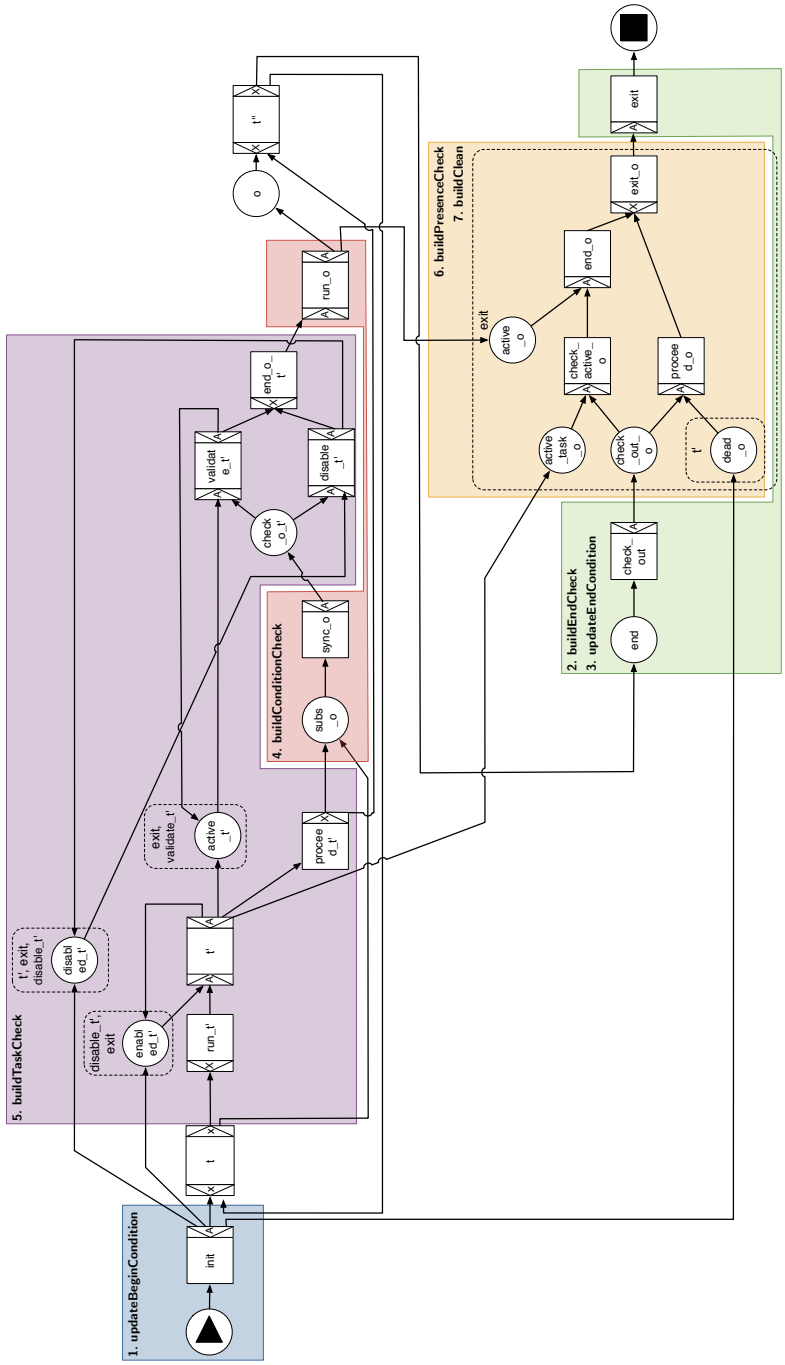


Figure 7.8 – Transformed FCW.

1. **Update begin condition.** The first step (blue area in Figure 7.8) adds an extra task ([init]) between the begin condition and the rest of the workflow. The goal of this task is to initialise the new places holding tokens necessary for the condition, tasks, and presence checks.

2. **Build end check.** The second step (green area in Figure 7.8) builds the necessary hooks for the task and condition checks. It is placed right before the end condition to make sure that the stop of the task has been activated before terminating the workflow.

3. **Update end check.** A new condition ((end)) is added to which all the other places connected to the original end condition are reconnected.

4. **Build condition check.** The fourth step (red area in Figure 7.8) builds the hook for the task check. Its purpose is to make sure that the task check is performed before the stop is activated.

5. **Build task check.** The task check (pink area in Figure 7.8) is structured so that if the task is activated once, then it must always be followed by the stop. In the example, [t'] activates (activate_t') , which enables [validate_t'] , and deactivates [disabled_t'] . If [validate_t'] has not been activated before the stop condition is reached, then the task is disabled ([disable_t'] removes the token from (enabled_t')), and the task ([t']) cannot be executed anymore. If the sequence tries to execute the task, it is deadlocked. This reveals a sequence that executes the task after the stop (violation of Definition 7.5.B). If (disable_t') is dead, it means that [t'] is mandatory.

6. **Build presence check.** The presence check (yellow area in Figure 7.8) verifies that if the task has been executed ((active_task_o) contains tokens), then its stop has also been activated ((active_o) contains tokens). If the task has been executed but not its stop, then [end_o] is deadlocked.

7. **Build clean.** Cancellation regions are added to remove the extra tokens created during the execution that could violate the proper completion condition. They are executed at the end of the workflow, before the end condition is reached.

The transformation thus allows to verify Definition 7.5.B, and collect the mandatory configuration tasks. To check Definition 7.5.A, we verify that the union of the features in the mandatory views ($V_{mandatory}$) completely covers the views.

The FCW in Figure 7.8 does not satisfy Definition 7.5.B. Indeed, [t'] can deadlock if (o) is executed before it, as in ρ_6. The deadlock is a result of the absence of token in (enabled_t') due to the execution of [disable_t']. The execution of [disable_t'] removes all the tokens in the places in its cancellation region, among which (enabled_t'). Another deadlock could occur in [end_o] if there is no token in (active_o). It means that [t'] has been executed but (o) has never been active before the end condition. This is a case similar to ρ_7. Definition 7.5.A is not satisfied either. Since [disable_t'] is not dead, [t'] is optional. Consequently, $V_{mandatory}$ is empty, and none of the features in v are propositionally defined by other views, as in ρ_5.

To sum up, the verification of the safety property is performed in four phases: (1) verification of the workflow soundness; (2) verification of the complete view coverage; (3) generation of a transformed workflow for each view and verification of the soundness property; (4) verification of the complete coverage of the mandatory views. This guarantees that at least one valid product can be derived from the FCW. The next stage is to ensure that the successive decisions of the users actually lead to a valid product.

7.7.2 Postponable decision

To verify that a decision can be postponed, we first have to compute the weak and strong dependency sets. These are typically computed prior to execution as they are independent of a particular sequence of execution.

The dependency sets of a task are computed backwards from the corresponding stop. Since we admit loops in the workflow, the dependency sets can evolve during the backward search. In the Spacebel case for instance, the backward search for the TMTC integrator from the stop condition (Final Satellite) generates the following sequence of places (simplified here for readability):

(loop) → TMTC Int. → Network Int. → System Engineer → (loop) → TMTC Int. → ...

When [TMTC Integrator] is first reached, $WD_{TMTC\ Integrator, Final\ Satellite}$ is empty because [(loop)] has no view attached to it. At the second time, the features in the views of the system engineer, network integrator and TMTC integrator are added to $WD_{TMTC\ Integrator, Final\ Satellite}$ because the loop iterates through them. The loop being infinite, we need to determine a condition that stops the iterations. That condition is that all the dependency sets must be stable, i.e., they do not evolve anymore. This means that we have to compute a fixed point over the dependency sets such that when the fixed point is reached, the dependency sets are complete.

The dependency set computation is implemented in Algorithm 1. The function WBFS performs a backward breadth-first search (BFS) of the workflow and

computes a fixed point over m'. m' is defined as the tuple $(m, wds, sds, \mathit{fixed})$ where: $m \in \mathcal{L}_{FCW}$; $wds : P \cup C \to \mathcal{P}(N)$ returns the weak dependency set; $sds : P \cup C \to \mathcal{P}(N)$ returns the strong dependency set; $\mathit{fixed} : T \cup C \to \mathbb{B}$ returns `true` if the dependency set of a place can still evolve, and `false` otherwise. For all the places p in the workflow, $sds(p)$ and $wds(p)$ are initialized to empty and $\mathit{fixed}(p)$ to false.

WBFS starts from condition o and initialises the queue Q of places to visit with the predecessors of o to bootstrap the backward search (Line 2). Line 3 sets the starting condition of the search to fixed. By setting it to fixed, we discard all the paths going out of the condition that could affect the places placed before the stop. Finally, lines 4 to 9 iterate until the dependency sets for each place stops varying, i.e., the fixed point is reached.

The UPDATEDS function first initialises the temporary weak (*tmpw*) and strong (*tmps*) dependency sets (lines 1 to 7). *tmpw* is set to $\emptyset$ to compute the union. Similarly, *tmps* is set to $\emptyset$ if it is a task with and *and*-split. Otherwise, it is set to N (global feature set) to compute the intersection. The values of *tmps* and *tmpw* are then computed (lines 8 to 20) as specified in Definition 7.8 and 7.9. Once done, the predecessors of p are enqueued to proceed with the backward search (lines 21 to 23). Finally, *wds* and *sds* are checked. If the sets are unchanged, then p is set to *fixed*, otherwise the dependency sets are updated (lines 24 to 29).

We prove below that the WBFS is complete, always terminates, and is correct.

Theorem 7.1 (Complete exploration of WBFS)

Given a sound m, WBFS *visits all the places that precede o.*

Proof. WBFS is a relaxed version of BFS as the places can be explored more than once. In WBFS, a place can be enqueued as long as the size of its dependency set varies (grows or shrinks). When it is not the case, it is marked has fixed and can no longer be searched. This amounts to marking the place as visited in the BFS algorithm. □

Theorem 7.2 (Termination of WBFS)

Given a sound m, WBFS *always terminates.*

Proof. We know from Theorem 7.1 that the exploration is complete. We know from line 24 in UPDATEDS that each place is visited while its dependency sets vary. The size of the dependency sets being bounded by $\emptyset$ and N and the stop being set to fixed before the exploration, they converge toward a stable value. When a place is fixed, it can no longer be enqueued and any occurrence still in the queue will no longer be visited. When all the places are fixed, the queue shrinks until it is empty. This guarantees that the loop (line 4) eventually terminates, and so does the algorithm. □

Algorithm 1 Computation of the dependency sets for all the views.

Require: m is (weak) sound

function WBFS(m',o) ▷ backward BFS of the workflow
  $Q \leftarrow \{q | q \in \bullet o\}$
  $\mathit{fixed}(o) \leftarrow$ TRUE ▷ prevent outgoing paths from affecting dependency sets
  while $Q \neq \emptyset$ **do**
    $p \leftarrow \mathit{pop}(Q)$
    if $\neg \mathit{fixed}(p)$ **then**
      UPDATEDS(m', p)
    end if
  end while
end function

function UPDATEDS(m',p) ▷ update the dependency sets
  if $p \in T \wedge \mathit{split}(p) =$ AND **then**
    $\mathit{tmps} \leftarrow \emptyset$
  else
    $\mathit{tmps} \leftarrow N$
  end if
  $\mathit{tmpw} \leftarrow \emptyset$
  for all $q \in p\bullet$ **do** ▷ compute temporary dependency sets
    if $q \in C$ **then**
      $\mathit{tmpw} \leftarrow \mathit{tmpw} \cup \mathit{wds}(q)$
      $\mathit{tmps} \leftarrow \mathit{tmps} \cap \mathit{sds}(q)$
    else
      $\mathit{tmpw} \leftarrow \mathit{tmpw} \cup (\mathit{wds}(q) \cup \mathit{start}^{-1}(q))$
      if $p \in T \wedge \mathit{split}(p) =$ AND **then**
        $\mathit{tmps} \leftarrow \mathit{tmps} \cup (\mathit{sds}(q) \cup \mathit{start}^{-1}(q))$
      else
        $\mathit{tmps} \leftarrow \mathit{tmps} \cap (\mathit{sds}(q) \cup \mathit{start}^{-1}(q))$
      end if
    end if
  end for
  for all $q \in \bullet p$ **do** ▷ enqueue the predecessors for the backward search
    $\mathit{push}(Q, q)$
  end for
  if $\mathit{wds}(p) = \mathit{tmpw} \wedge \mathit{sds}(p) = \mathit{tmps}$ **then** ▷ check for changes
    $\mathit{fixed}(p) \leftarrow \mathit{TRUE}$ ▷ the dependency sets of p are stable
  else
    $\mathit{wds}(p) \leftarrow \mathit{tmpw}$ ▷ the dependency sets of p have changed
    $\mathit{sds}(p) \leftarrow \mathit{tmps}$
  end if
end function

Theorem 7.3 (Correctness of WBFS)

Given a sound m, the weak and strong dependency sets of places that precede o computed by WBFS *respectively satisfy Definitions 7.8 and Definitions 7.9.*

Proof. We know from Theorem 7.1 that all the places are visited, and that they can only be marked has fixed when their dependency sets are stable. Also, the BFS guarantees that all the places between p and o are explored before p. Any change to one of them is thus automatically propagated to p. We have to demonstrate that:

1. *wds* are correct. If it was possible to find a feature f that should be in $wds(p)$ but that is not, it would mean that $\exists p_j \bullet p \xrightarrow{*} p_j \xrightarrow{*} o$ such that $f \in wds(p_j)$ but $f \notin wds(p)$. If $p \rightarrow p_j$, we have $f \in wds(p)$ by definition of the union. If $\exists p_i \bullet p \xrightarrow{*} p_i \rightarrow p_j$, then, again by definition of the union, we have $f \in wds(p_i)$. Inductively, it is not possible that $wds(p)$ does not contain f.

2. *sds* are correct. We have to prove that (1) $\nexists f \notin sds(p)$ that should have been propagated to p, and (2) every $f \in sds(p)$ has been correctly propagated to p. Let us take $p_j \bullet p \xrightarrow{*} p_j \xrightarrow{*} o$. If $p \rightarrow p_j$ and $split(p) = AND$, then $f \in sds(p)$ by definition of the union. If $split(p) = OR \vee XOR$, then $f \in sds(p)$ iff $f \in \bigcap_{p_s \in p\bullet} sds(p_s)$.

   If $p \xrightarrow{+} p_j$, then for all $p_p \in \bullet p_j$ such that $split(p_j) = AND$ we have $f \in sds(p_p)$ by definition of the union. If $split(p_j) = OR \vee XOR$, then $f \in sds(p_p)$ iff $f \in \bigcap_{p_s \in p_p\bullet} sds(p_s)$. Inductively, $sds(p)$ will contain f iff it is propagated by the places in F_{p,p_j}.

□

Once computed, dependency sets are used at execution time to evaluate the propositional definability of features left open in a view. The strong and weak definability of a view v with task t and stop o are computed by checking that all the open features f respectively satisfy $pdefines(SD_{t,o}, f)$ and $pdefines(WD_{t,o}, f)$. Features that do not satisfy either of these tests must be decided upon before leaving the ongoing task.

7.8 Experiments

In Section 6.4, we have seen how our algorithms satisfy the properties defined in Section 7.6. These algorithms have been implemented in our FCW toolset. Details on the implementation are available in Chapter 9. The research questions addressed by our experiments are:

RQ7.1 *How efficient are our algorithms?* To verify satisfiability, our algorithm relies on the verification of the soundness of the transformed workflow. We measure the performance of the verification. To compute dependency sets, we propose a fixed point algorithm. We evaluate how fast it converges to a solution.

RQ7.2 *How workflow constructs affect the performance of our algorithms?* Some constructs in a workflow are known to severely affect the performance of reasonings [WVvdA+09]. In particular, we want to know how loops and *or*-joins affect the soundness analysis of the transformed workflow and the computation of dependency sets.

These two questions aim at evaluating Algorithms 1 and 2. Since view coverage, workflow soundness, and propositional definability have already been evaluated elsewhere, the experiments focus on our contribution.

To answer these research questions, we engineered 52 FCWs from 13 workflows and 2 FMs. These FCWs were built to (1) evaluate a wide range of cases with complementary levels of complexity, and (2) highlight the limitations of our algorithms.

The 13 workflows were collected from 4 different projects: YAWL's test set (W1,W3,W5-7); YAWL4ProductRecall[2] (W9); YAWL4Film[3] (W10); a municipality project containing four processes for birth, death, marriage, and unborn children regulation (W8,W11-13); the CFDP case (W4); the example in Figure 7.6 (W2). These workflows count between 5 and 51 elements.

Each of these workflows was normalised as follows. First, we verified their soundness. The unsound workflows (6 in total) were altered to meet the soundness pre-condition of our algorithms. In addition, we also removed YAWL constructs that are not supported by our approach. The following rules were applied. Composite and multiple instance tasks are transformed into regular tasks. Every configuration task in a cancellation region is removed from that cancellation region. In total, 16 composite tasks were transformed affecting 8 workflows, 3 multiple instance were transformed affecting 3 workflows, and no cancellation region was transformed.

Finally, every workflow was duplicated, and loops were manually added (respectively removed) to measure their impact on performance.

Next, each of these 26 workflows was linked to two FMs to build our test base of 52 FCWs. To measure the performance of the algorithms under realistic settings, we use two different variability models as input: the Linux Kernel (version 2.6.28) and the vmWare hardware platform of eCos [BSL+10, SLB+11]. These models respectively count 5701 and 1244 features, and are among the

[2] `http://www.yawlfoundation.org/casestudies/productrecall`
[3] `http://www.yawlfoundation.org/casestudies/yawl4film`

largest models publicly available[4]. Between 1 and 10 views were randomly assigned to each workflow according to its size, and the features of each model were divided evenly between them.

The soundness analyses were executed on YAWL 2.1 with the YAWL and Reset reduction rules optimizations enabled. The workflow transformation and dependency set algorithms were both implemented in Java v1.6. The computer used for the evaluation was a MacBook Pro running Mac OS 10.6 with an Intel Core 2 Duo at 2.8GHz and 4GB of RAM.

7.8.1 Transformed workflow analysis

The results on the soundness analyses are shown in Figure 7.9. Since this analysis is independent on the FM linked to it, we only report here the measures for the 26 workflows. The left Y-axis denotes the number of elements (continuous line) and views (dashed line) in the workflow. The right Y-axis denotes the time in seconds (seconds). For each workflow (X-axis), the graph reports the total average time needed to complete the soundness analysis for all the views of the workflow transformed by Algorithm 2, with and without loops. To minimise the bias introduced by concurrent processes running on the operating system, the time was calculated over three successive executions of the soundness analysis for each view. The values reported in the graph are the sum of the averages. Bars marked with a **X** indicate approximate results returned by the analysis because it could not explore enough markings.[5] Finally, W3, W5, W7 and W9 contain *or*-joins.

Let us highlight the key elements of Figure 7.9 that answer RQ7.1. First, apart from the loop version of W12 whose analysis was manually stopped, the analysis time is always under 500 seconds. That result is mostly a consequence of the limit on the number of markings imposed by YAWL. Note that successful analyses provide a result in less than 350 seconds. Secondly, the number of elements in a workflow does not have a significant impact on processing times. W10 for instance, has almost three times more elements than W4, and has an average processing time of 43 seconds compared to 51 seconds for W4. A similar conclusion can be drawn for the number of views. W11, for instance, contains 7 views while W1 contains only one. Their respective average processing time are 2 seconds and 12 seconds. Although W11 contains 7 times more views and about 6 times more elements, it is 6 times faster to analyse. Consequently, the size of an FCW does not explain discrepancies between the results.

[4] `http://code.google.com/p/linux-variability-analysis-tools/source/browse/?repo=formulas`

[5] YAWL's engine is configured to stop the analysis when the number of reachable markings grows beyond 5000. Since the number of markings is potentially infinite, that bound aims *"to balance responsiveness and precision"* [WVvdA+09].

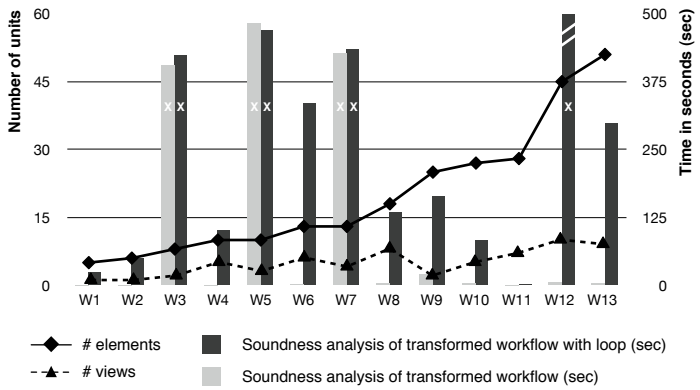


Figure 7.9 – Results of the soundness analysis for the transformed workflows.

To better understand that problem and answer RQ7.2, we investigate the other differentiating marker: workflow constructs. As reported in [WVvdA+09], *or*-joins and loops have a detrimental effect on performance. In most cases (W3, W5, and W7), *or*-joins caused an explosion of the state space, which does not allow the analysis to complete. In these particular cases, loops have a marginal impact on processing time; all the complexity is induced by *or*-joins. However, the impact of loops alone clearly stands out in Figure 7.9. The average processing time for workflows without loops that do not fail (i.e. W3, W5, and W7) is 4 seconds, whereas for workflows with loops that do not fail (i.e. W3, W5, W7, and W12) is 132 seconds. Clearly, *or*-joins have a significantly higher chance to max out the exploration limit of the soundness analysis (75%) than loops alone (8%).

An important information, not reported in Figure 7.9, is that the verification of the original workflows is orders of magnitude faster than the results shown in the graph. To explain that variation, we study workflow constructs *and* the mapping of views on tasks.

The individual study of each transformed workflow produced for a view (not reported in the graph) reveals that the average time for a view that is not part of a loop is below one second. W11 is a perfect example of that. None of the 7 tasks mapped to views are involved in a loop, which results in extremely low processing times. The soundness analysis of configuration tasks in a loop jumps to several dozen seconds. That leap, however, only affects the correctness of the results for one workflow: W12. The problem is that two views are mapped to tasks that are in embedded concurrent loops. These two tasks are responsible for the leap in processing time. All the others complete correctly.

or-joins expose a similar pattern. Tasks placed before a split whose outgoing paths are later joined with an *or* have a computation time below one second. The same holds for the task with the *or*-join and those placed after it. That explains why the processing time of W9 is reasonable despite the presence of an *or*-join. The computation for tasks between the split and the *or*-join is well beyond 150 seconds, and the results are most of the time approximate; irrespective of the complexity of the workflow. For instance, W3 and W7 have different levels complexity but fail likewise.

These results clearly show that *or*-joins degrade most the accuracy of the results. In fact, *or*-joins require partial synchronisation of all the active paths in the workflow. This synchronisation is the bottleneck in the analysis because the reasoning engine waits "*until it is not possible to add any relevant tokens to the set of input condition*" [vdAtH05], causing an explosion of the exploration space. The accuracy of our algorithm can be improved at the expense of one syntactical restriction: configuration tasks should not be placed on paths synchronised at an *or*-join.

7.8.2 Dependency sets

The evaluation of Algorithm 1 was conducted separately for the Linux Kernel and eCos. Figure 7.10 shows the processing time in seconds of the algorithm for these models and the different workflows, with and without loops. Each value reported in the graph is an average of three successive runs of the algorithm.

Starting with RQ1, the first striking element is the difference between the Linux Kernel and eCos. The maximum computation time for eCos is $0,9$ second, whereas for the Linux Kernel it is 17 seconds. Since the algorithm only considers the number of features irrespective of the complexity of the FM, the size of the FM is clearly the major performance predictor. To a lesser extent, the number of elements in the workflow also influences performances. The graph also shows that these variations are exacerbated by the number of features to process. Both eCos and the Linux Kernel exhibit the same variations but differences are amplified in the latter case.

Unlike for safety, workflow constructs have a marginal impact on performance. Loops induce slightly higher processing times on average: $4,59$ vs $5,68$ for Linux ; $0,23$ vs $0,3$ for eCos. *or*-joins do not affect performance since Definition 7.8 is independent of the type of join, and Definition 7.9 treats *or* and *or*-joins similarly. The peaks of W6 and W9 do not seem to be due to a particular pattern. They are peculiarities of the design of the workflow and view-to-task mapping. The same holds for the higher processing time of the non-loop version of W11.

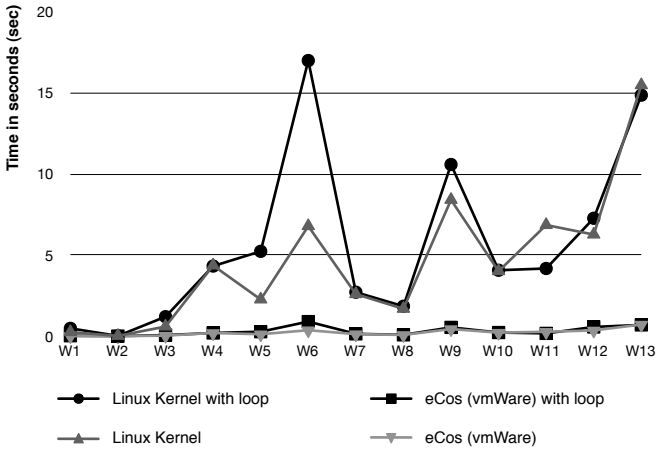


Figure 7.10 – Results of the dependency set analysis for the Linux Kernel and eCos (vmWare) FMs.

7.9 Threats to validity

The experiments conclude our technical contribution. We now discuss the limitations of our experiments, our assumptions on workflow constructs, and the coverage of mandatory tasks.

Experiments. To evaluate the performance of our algorithms, we engineered FCWs from two FMs and 13 workflows to simulate large-scale projects. We chose to build these models because FCWs are still an emerging technology, and the models available are rather simple. To evaluate the performance of our algorithms, we needed larger and more complex models. Similar approaches have also been applied to evaluate new reasoning techniques for which complex models were not available (e.g., [MWC09]).

We mitigate the threat of arbitrary FCW creation by using actual FMs and workflows. Both FMs are currently used in full-blown open source applications. The workflows count between 5 and 51 elements, which is representative of the current practice in business process (workflow) modelling. First, because the configuration process is usually embedded in a more complex process or is itself a high-level process whose tasks are decomposed into dedicated sub-processes. Secondly, according to [Gil10], about 75% of the business processes used in industry are rather *"simple"*; irrespective of the size of the company.[6]

[6]Gilbert [Gil10] reports that these workflows are "*so simple that they are usually ex-*

Finally, as the experiments show, the workflow size is not the leading factor of complexity. Yet, additional case studies are needed to ensure that the decisions we made to build these FCWs unveiled all the strengths and weaknesses of our algorithms.

Propositional definability has not been evaluated in our experiments. Yet, it is a crucial part of most our algorithms. We recalled in Section 7.7.1 that propositional definability can be obtained with a single SAT check. To date, the largest FMs count up to 6000 features with thousands of constraints [BSL+10]. We know from [MWC09] that a SAT solver can verify the satisfiability of such models in a few hundred milliseconds for the larger models. That fact has also been confirmed by our experience in FM analysis. Therefore, for any given dependency set, the verification at execution time of open features is unlikely to degrade the response time of an interactive configurator.

Unsupported constructs. In Section 7.3, some YAWL constructs were excluded from our definition: cancellation regions, composite tasks, and multiple instance tasks. Cancellation regions were discarded because they could cancel an ongoing configuration task. Whether the configuration should be part of a transaction and rolled back to its previous state, or the decisions made should be preserved is still unclear. Similarly, how composite configuration tasks and configuration tasks with multiple occurrences should be handled requires further investigation. Our experience with FCWs never required such constructs but additional empirical evaluation is necessary.

Complete coverage of mandatory tasks. Definition 7.5.A assumes that mandatory configuration tasks must completely cover the set of features. The benefit is that whatever path is followed at execution time, the final configuration will always be complete. The flip side of the coin is that optional configuration tasks can only provide alternative ways of reaching a solution without bringing in new features.

Alternatively, which optional tasks must be executed given the current decisions could be computed at execution time. Such analysis relies on dependency sets and open features in the complete model. Our algorithms can already provide that information. The challenge is the analysis of possible paths in the workflow. Such an approach puts a heavier burden on execution time analyses, which could harm the responsiveness of the configurator.

Finally, default values, and semi or fully automated completion mechanisms [JBGMS10] could be used at the end of the process to complete the configuration. These are less resource-demanding since no analysis of the configuration process is required. The downside of default values, though, is that their assignment at design time can become very cumbersome for large models.

---

changed via emails" (*sic.*).

7.10 Related Work

This section revisits some selected work on workflow configuration and the integration of processes with variability models.

Gottschalk *et al.* [GvdAJVLR07] developed C-YAWL, a configurable version of YAWL, and a tool that allows the configuration and generation of YAWL workflows. La Rosa *et al.* [LRvdADTH08] also focus on workflow configuration. They propose a questionnaire-based approach to resolve variability in the process. They also define partial and full dependencies to control the order in which questions are asked. Their approach is complementary to ours and can be used upstream of FBC to tailor the configuration process for a particular application.

Rabiser *et al.* [RGD07] propose an approach supporting product configuration based on *decision models*. Essentially, decision models represent *assets* (e.g., features) tied to *decisions*, bound together through logic dependencies. Decisions stand for the intervention of a *role* selecting assets during product configuration. Decisions, roles and assets are thus all part of a single decision model. They also discuss how models need to be prepared to meet the requirements of a specific project before allowing product derivation. Decision models differ from FCWs in that the configuration process is entangled within the decision model/questionnaire. By separating the workflow from the options, we argue that FCW achieves better separation of concerns between process and decision making. However, FCWs and decision models/questionnaires are complementary. The control provided by dependencies could offer fine-grained scheduling within each view.

White *et al.* [WBDS09] reason about contextual constraints (e.g., yearly budget) to schedule the configuration of an FM into multiple steps. They also show how the mapping of the FM and constraints in CSP allows deriving valid configurations. That latter approach is complementary to ours as constraints on steps could be used during the validation of the workflow at design time.

Mendonça *et al.* [MCMdO08] partition an FM into configuration spaces whose configuration order is captured in a configuration plan. The order is determined by a dependency analysis between the different configuration spaces. While their approach is fully automated, ours enables a more flexible definition of concerns and precedence among views.

Acher *et al.* [ACLF10] use FMs to capture the variability of web services that are threaded together in a workflow. However, contrary to FCWs, they do not assume the pre-existence of a global FM. They define composition operators that merge FMs of "*connected*" services. The two main differences with FCWs are that they do not support crosscutting constraints, and the workflow does not contain configuration tasks but provided services.

7.11 Chapter Summary

This chapter introduced a new formalism called FCW allowing to organise interrelated views as part of an unambiguous configuration workflow. For this formalism, we proposed a semantics that builds upon MLSC and YAWL.

The primary benefit of FCWs is that they allow explicit modelling of non-trivial configuration processes, thereby overcoming the original limitations of MLSC and bringing assistance to the product management. From the resource allocation perspective, FCWs facilitate the task assignment to the different roles played by the stakeholders. From the control standpoint, stops of an FCW provide milestones for the project manager and keep him informed about the evolution of the configuration process, whereas feedback loops allow to define synchronisation points among roles.

However, when using FCWs, design defects and unguided configuration can lead to inconsistent or incomplete products. To tackle that problem, this chapter has extended joint work on FM and workflow analysis by:

- *Formally defining the concepts of satisfiability and postponable decision.* These definitions build upon the known properties of view coverage of FMs and workflow (weak) soundness. These properties have been extended with FCW safety, and weak and strong dependency sets.

- *Providing reference algorithms for their verification.* Our algorithms use SAT solvers and the YAWL reasoning engine. We have shown how the soundness analysis of a transformed workflow can (1) verify the safety of an FCW, and (2) identify mandatory configuration tasks. The computation of dependency sets required to test postponable decisions is achieved by a fixed point algorithm.

- *Evaluating the performance of our algorithms and the impact of workflow constructs.* 52 FCWs have been used to evaluate the performance of safety analysis and dependency set computation. In the former case, *or*-joins turned out to have a detrimental effect on computation time and accuracy. Although affecting computation time, loops only affected the safety analysis of one workflow. In the latter case, the number of features in the FM had a noticeable impact on processing time but not on correctness.

This chapter ignored possible conflictual decisions between views. Conflict detection and resolution is studied in the next chapter. The implementation of FCW modelling, execution and verification in our toolset is presented in Chapter 9.

Chapter

8 Towards Conflict Management

8.1 Open Issues

Until now, we have assumed that user decisions are invariable during the configuration process. Yet, change is inherent to configuration. Most configurators implement some form of assistance to deal with change and help users reach a correct and complete configuration. To do so, they detect possible configuration errors, report, and avoid them. A configuration error is a decision that conflicts with some constraints. Satisfying these constraints is often non-trivial. Variability languages often carry advanced constructs that introduce hidden constraints, and constraint rules declared in different places of the variability model may have interactions. The interplay of these factors often leads to very complex situations.

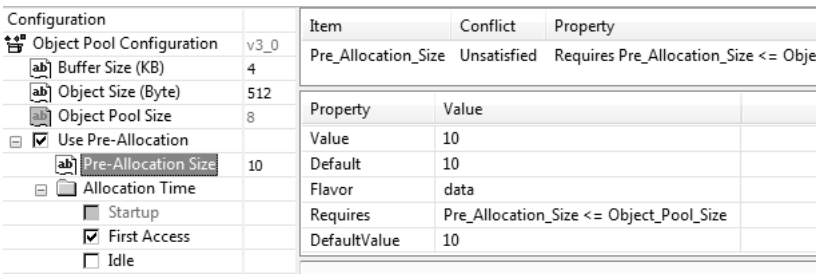


Figure 8.1 – The eCos Configurator

Some configuration tools, like those based on Kconfig (the Linux kernel

configuration description language), implement an error avoidance mechanism that automatically deactivates an option when a certain constraint is violated. Inactive options are no longer available to the user unless the constraint is satisfied again. Other configurators, like the eCos configurator and its configuration description language CDL (Figure 8.1), add an interactive resolution mechanism on top of the avoidance mechanism. This approach allows violating some constraints, but proposes a fix for each violated constraint. A fix denotes a set of changes that would restore the consistency of the current configuration.

To better understand what challenges are faced by the users of modern configurators, we carried out two empirical studies of Linux and eCos. Two questionnaires were submitted to forums, mailing lists and experts with whom we collaborate. In total, we collected answers from 97 Linux users with up to 20 years of experience, and 9 eCos users with up to 7 years of experience. The full report of this study is available as a technical report [HXC11], and a synthesis of the results is presented in [HXC12]. We present here the two challenges that stand out most from this study and that are addressed in this chapter:

Activating inactive features. 20% of the Linux users report that, when they need to change an inactive option, they need at least a "few dozen minutes" in average to figure out how to activate it. 56% of the eCos users also consider the activation of an inactive option to be a problem.

Fix incompleteness. Existing configurators generate only one fix for an error. However, there are often multiple solutions to resolving an error, and the user may prefer other solutions. 7 out of 9 eCos users have encountered situations where the *generated fix is not useful*. That claim is corroborated by Berger *et al.* [BSL+10] who report that eCos users complain about the incompleteness of fixes on the mailing list.

Since we also need to satisfy the corresponding constraint to activate a feature, activation is inherently the same as resolving a configuration error, and the idea of fixes would also work for activation. As a result, a possible solution for the above two problems is to generate fixes for both resolving errors and activating features, and the list fixes should be complete so that the user can choose the one he wants.

To achieve this goal, two main challenges need to be addressed. First, a previous study of eCos models [PNX+11] shows that non-Boolean operators, such as arithmetic, inequality, and string operators, are quite common in their constraints. In fact, the models contain four to six times more non-Boolean constraints than Boolean ones. Non-Boolean constraints are challenging since there is often an infinite number of ways to satisfy them. Computing such infinite list of fixes is pointless. Thus, a *compact* and *intensional* representation of fixes is needed. Second, many existing approaches rely on constraint solvers

to generate fixes, either using solvers for MAXSAT/MAXSMT [JM11] or the optimizing capability of CSP solvers [WSB+08]. However, all these solvers return only one result per call, which is not easily applicable to the generation of complete lists of fixes. A new method to generate complete lists of fixes still needs to be found.

This chapter relaxes the strong hypothesis that FMs are entirely Boolean to fully address the configuration challenges in operating systems. Our contribution is threefold:

Range fixes. We propose a novel concept, *range fix* (Section 7.2),to address the first challenge. Instead of telling users what concrete changes should be made, a range fix tells them what options should be changed and in what range the value of each option can be chosen. A range fix can represent infinite number of concrete fixes and still retains the goal of assisting the user to satisfy constraints. Particularly, we discuss the desired properties of range fixes, which formalize the requirement of the fix generation problem. In addition, we also discuss how constraint interactions should be handled in our framework (Section 8.5).

Fix generation algorithm. We designed an algorithm that generates range fixes automatically (Section 8.4) to address the second challenge. Our algorithm builds upon Reiter's theory of diagnosis [Rei87, GSW89] and SMT solvers [DMB08]. Additionally, our algorithm is designed for a general representation of constraints and variables, which makes it potentially useful in other areas such as debugging.

Evaluation with eCos. Our algorithm is (1) applied on eCos CDL (Section 8.6) and (2) evaluated on data from five open source projects using eCos (Section 8.7). The evaluation compares three different fix generation strategies and concludes that the propagation strategy is the most effective one on our dataset. Specifically, for a total of 117 constraint violations, the evaluation of the propagation strategy shows that our notion of range fix leads to mostly simple yet complete sets of fixes (83% of the fix lists have sizes smaller than 10, where the size is measured by summing up the number of variables in all the fixes in the list). It also demonstrates that our algorithm can generate fixes for models containing hundreds of options and constraints in an average of 50ms and a maximum of 245ms.

Finally, we discuss threats to validity in Section 8.8, the related work in Section 8.9, and conclude in Section 8.10 with two issues on the application of range fixes to collaborative FBC.

8.2 Working Example: eCos

We motivate our work with a concrete example based on the eCos configurator [VD01]. Figure 8.1 shows a small model for configuring an object pool. The left panel shows a set of options that can be changed by the user, organized into a tree. The lower-right panel shows the properties of the current option, defined according to the eCos models. Particularly, the *flavor* property indicates whether the option is a Boolean option or a data option. A Boolean option can be either selected or unselected; a data option can be assigned an integer or a string value. In Figure 8.1, "Pre-Allocation Size" is a data option; "Use Pre-Allocation" is a Boolean option.

Besides the flavor, each option may also declare constraints using *requires* property or *active-if* property. When a requires constraint is violated, an error is reported in the upper-right panel. In Figure 8.1, option "Pre-Allocation Size" declares a requires constraint demanding its value be smaller than or equal to "Object Pool Size"; and an error is reported because the constraint is violated.

An active-if constraint implements the error avoidance mechanism. When it is violated, the option is disabled in the GUI and its value is considered as zero. Figure 8.2 shows the properties of the "Startup" option. This option declares that at most half of the object pool can be pre-allocated. Since this constraint is violated, the "Startup" option is disabled, and the user cannot change its value.

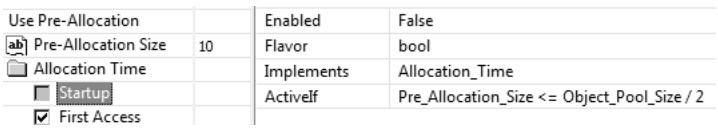


Figure 8.2 – Option "Startup"

Fixing a configuration error or activating an option requires satisfying the corresponding constraints. In order to fix the error on "Pre Allocation Size" in Figure 8.1, we need to look up the definition of "Object Pool Size". In Figure 8.3, we see that "Object Pool Size" declares a *calculated* property meaning that the value of the option cannot be modified by the user. Instead, it is determined by a declared expression. As a result, the constraint declared on "Pre-Allocation Size" is, in fact, the following:

```
Pre_Allocation_Size <= Buffer_Size * 1024 / Object_Size
```

Furthermore, according to the CDL semantics, a constraint is effective—and thus considered by the error checking system—only when its containing option is active. An option is active only when its active-if constraint is satisfied and its parent option is selected. "Pre-Allocation Size" has a parent, yielding the following complete constraint:

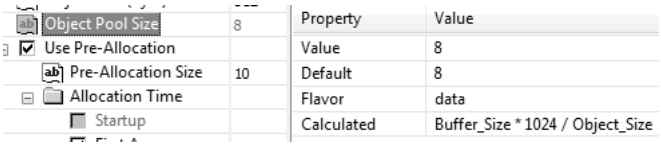


Figure 8.3 – Option "Object Pool Size"

```
Use_Pre_Allocation -> (Pre_Allocation_Size <=
   Buffer_Size * 1024 / Object_Size)
```

By analyzing the constraint, we realize that we may fix the error by one of the following changes: decreasing "Pre-Allocation Size", or increasing "Buffer Size", or decreasing "Object Size", or, more simply, disabling the pre-allocation function. Now we could choose one of these possibilities and navigate to the respective option to make the change.

This example shows that there are three sub-tasks for enabling a constraint. First, the user needs to figure out the complete constraint according to the constraint language. Since variability languages often have fairly complex semantics on visibility and value control [BSL+10], it is very easy to overlook some part of the constraint. Secondly, users need to analyse the constraint and figure out how to change the options to make it satisfied. In practice, constraints can be very large. One constraint we have found in a CDL model contains 55 options references and 35 constants, connected by 66 logical, arithmetic and string operators. It is very difficult to analyse such a large constraint. Thirdly, users have to navigate to the corresponding options and make the changes. Real world variability models contain thousands of options, e.g., an eCos model was reported [BSL+10] to contain 1244 options, which makes navigation very cumbersome [HXC11].

8.3 Range Fixes

8.3.1 Overview of the solution

Our approach automatically generates a list of range fixes to help satisfy a constraint. For the error in Figure 8.1, we will generate the following fixes.

- `[Use_Pre_Allocation := false]`
- `[Pre_Allocation_Size:  Pre_Allocation_Size <= 8]`
- `[Buffer_Size:  Buffer_Size >= 5]`
- `[Object_Size:  Object_Size <= 4096]`

Each range fix consists of two parts: the option to be changed and a constraint over the options showing the range of values. The first range fix is also a concrete assignment, and will be automatically applied when selected.

The other fixes are ranges. If the user selects, for example, the second fix, the configurator will highlight option "Pre-Allocation Size", prompt the range "`<=8`", and ask the user to select a value in the range.

8.3.2 Definitions

Although feature modelling languages have different constructs and semantics, existing work [SHTB07, CW07, BS10a, BSL+10] shows that they basically boil down to a set of variables (options) and a set of constraints. Our approach also builds upon this principle.

In essence, a feature modelling language provides a universe of typed variables $\mathbf{V}$, and a constraint language $\Phi(\mathbf{V})$ for writing quantifier-free predicate logic constraints over $\mathbf{V}$. Consequently, a constraint violation can be defined as follows.

Definition 8.1 Constraint violation

A constraint violation *consists of a tuple* (V, e, c)*, where* $V \subseteq \mathbf{V}$ *is a set of typed variables; the current configuration* e *is a function assigning a type-correct value to each variable; and* $c \in \Phi(V)$ *is a constraint over* V *violated by* e*.* ■

A fix generation problem for a violation (V, e, c) is to find a set of range fixes to help users produce a new configuration e' such that c is satisfied, denoted as $e' \models c$.

Consider the following example of a constraint violation:

$$\begin{array}{lcl} V & : & \{m : \texttt{Bool}, a : \texttt{Int}, b : \texttt{Int}\} \\ e & : & \{m = \texttt{true}, a = 6, b = 5\} \\ c & : & (m \rightarrow a > 10) \wedge (\neg m \rightarrow b > 10) \wedge (a < b) \end{array} \tag{8.1}$$

All range fixes we have seen so far change only one variable, but more complex fixes are sometimes inevitable. For example, we cannot solve violation (8.1) by changing only one variable. Several alternative fixes are possible:

- $[m := \texttt{false},\ b : b > 10]$
- $[(a, b) : a > 10 \wedge a < b]$

The first fix contains two parts separated by ",", each changing a variable. We call each part a *fix unit*. The second fix is more complex. This fix contains only one fix unit, but the range of this fix unit is defined on two variables. When the fix is executed, the user has to choose a value for each variable within the range.

Taking the above forms into consideration, we can define a range fix. A *range fix* r for a violation (V, e, c) is a set of fix units. A fix unit can be either an *assignment unit* or a *range unit*. An assignment unit has the form of "$var := val$" where $var \in V$ is a variable and val is a value conforming to the type of var. A range unit has the form of "$U : cstrt$", where $U \subseteq V$ is a set of variables and $cstrt \in \Phi(U)$ is a satisfiable constraint over U specifying the new ranges of the variables. A technical requirement is that the variables in fix units should be disjoint, otherwise two different values may be assigned to one variable.

We use $r.V$ to denote the set of the variables to be changed in all units. We use $r.c$ to denote the conjunction of the constraints from all units. The constraint from an assignment unit "$var := val$" is $var = val$, and the constraint from a range unit "$U : cstrt$" is $cstrt$. For example, let r be the range fix $[m := \texttt{false},\ b : b > 10]$, then $r.V = \{m, b\}$ and $r.c$ is $m = \texttt{false} \wedge b > 10$.

Applying range fix r of violation (V, e, c) to e will produce a new configuration interactively. We denote all possible configurations that can be produced by applying r to e as $r \triangleright e$, where $r \triangleright e = \{e' \mid e' \models r.c \wedge \forall v \in V(e'(v) \neq e(v) \rightarrow v \in r.V)\}$

8.3.3 Desired Properties

A simple way to generate a fix from a violated constraint is to produce a range unit where the changed variables are all variables in this constraint and the range of these variables is the constraint itself. For example, the fix for violation (8.1) could be $[(m, a, b) : (m \rightarrow a > 10) \wedge (\neg m \rightarrow b > 10) \wedge (a < b)]$. However, such a fix is as difficult to understand as the original constraint. In this subsection, we discuss the desired properties of range fixes.

Suppose r is a range fix for a violation (V, e, c). The first desired property is that a range fix should be correct: all configurations that can be produced from the fix must satisfy the constraint.

Property 8.1 **Correctness**

$\forall e' \in (r \triangleright e),\ e' \models c$

■

Each value currently assigned to a variable is a configuration decision made by the user, and a fix should alter as few decisions as possible. The second desired property is thus that a fix should change a minimal set of variables. For example, $[m := \texttt{true},\ b : b > 10]$ is preferable to $[m := \texttt{true},\ b : b > 10,\ a : a = 9]$ because the latter unnecessarily changes a, which does not contribute to the satisfaction of the constraints.

Property 8.2 **Minimality of variables**

There exists no fix r' for (V, e, c) such that r' is correct and $r'.V \subset r.V$.

■

Minimal fixes, however, might not cover all possible changes that resolve a violation, and these uncovered cases might be preferred by some users. However, as our evaluation will show, minimality is good heuristics in practice.

Thirdly, after determining a set of variables, we would like to present the maximal range of the variables. The reason is simple: extending the range over the same set of variables gives more choices, and usually neither decreases readability nor affects more existing user decisions. For example, $[m := true,\ b : b > 10]$ is better than $[m := true,\ b : b > 11]$ because it covers a wider range on b.

Property 8.3 **Maximality of ranges**

There exists no fix r' for (V, e, c) such that r' is correct, $r'.V = r.V$ and $(r \triangleright e) \subset (r' \triangleright e)$.

■

Fourthly, after deciding the range over the variables, we would like to represent the range in the simplest way possible. Thus, another desired property is that a fix unit should change as few variables as possible. In other words, no fix unit can be divided into smaller equivalent fix units. We call this property *minimality of units*.

However, as we treat Φ as a general notion, our generation algorithm cannot ensure all fix units are minimal. Therefore we do not treat this property as part of the formal requirement. However, this does not seem to be a limitation in practice; all fix units generated in our evaluation contain only one variable, which are minimal by construction.

Armed with these properties of range fixes, we can define the *completeness* of a list of fixes. Since the same constraint can be represented in different ways, we need to consider the semantic equivalence of fixes. Two fixes r and r' are *semantically equivalent* if $(r \triangleright e) = (r' \triangleright e)$, otherwise they are *semantically different*.

Property 8.4 **Completeness of fix lists**

Given a constraint violation (V, e, c), a list of fixes L is complete iff

- *any two fixes in L are semantically different,*
- *each fix in L satisfies Property 8.1, 8.2, and 8.3.*
- *and any fix that satisfies Property 8.1, 8.2, and 8.3 is semantically equivalent to a fix in L*

■

Thus, a *fix generation problem* is to find a complete list of fixes for a given constraint violation (V, e, c).

8.4 Fix Generation Algorithm

In Section 8.3.3 we claimed that a fix should change a minimal set of variables and have a maximal range. As a result, our generation algorithm consists of three steps. (i) We find all minimal sets of variables that need to be changed. For example, in violation (8.1), a minimal set of variables to change is $D = \{m, b\}$. (ii) For each such set of variables, we replace any unchanged variable in c by its current value, obtaining a maximal range of the variables. In the example, we replace a by 6 and get $(m \to 6 > 10) \wedge (\neg m \to b > 10) \wedge (6 < b)$. (iii) We simplify the range to get a set of minimal, or close to minimal, fix units. In the example we will get $[m := \texttt{true},\ b : b > 10]$. Step (ii) is trivial and does not demand further developments. We now concentrate on steps (i) and (iii).

8.4.1 From constraint and configuration to variable sets

To collect all minimal variable sets, we resort to Reiter's theory of diagnosis [Rei87]. This theory defines the problem of diagnosis, and gives an incomplete algorithm for solving the problem. This algorithm was later corrected by Greiner *et al.* [GSW89] and is now known as *HS-DAG* algorithm. Fundamentally, Reiter's theory assumes constraint sets that can be split into *hard* and *soft* constraints. The set of hard constraints is invariable and assumed satisfiable. The set of soft constraints can be altered and also be unsatisfiable. A *diagnosis* is a subset of soft constraints that, when removed from the set, restores the satisfiability of the whole set. The problem of diagnosis is to find all minimal diagnoses from a set of hard and soft constraints.

Given the constraint violation (V, e, c), we convert the problem of finding minimal variable sets to the problem of diagnosis by treating c as a hard constraint and converting e into soft constraints. For example, violation (8.1) can be converted into the following constraint set.

Hard constraint (c):
[0] $(m \to a > 10) \wedge (\neg m \to b > 10) \wedge (a < b)$
Soft constraints (e):
[1] $m = \texttt{true}$
[2] $a = 6$
[3] $b = 5$

To make the whole set satisfiable, we need to remove at least constraints $\{1, 3\}$ or constraints $\{2, 3\}$, which correspond to two variable sets $\{m, b\}$ and $\{a, b\}$.

To find all diagnoses, Reiter's theory uses an ability of most SAT/SMT solvers: finding an unsatisfiable core. An *unsatisfiable core* is a subset of the soft constraints that is still unsatisfiable. For example, the above constraint set has two unsatisfiable cores $\{1, 2\}$ and $\{3\}$.

If we cancel a constraint from each unsatisfiable core, we get a diagnosis. The HS-DAG algorithm implements this idea by building a DAG, such that each node is labelled either by an unsatisfiable core or SAT, and each arc is labelled by a constraint that is cancelled. The union of the labels on every path from the root to a SAT node defines a diagnosis.

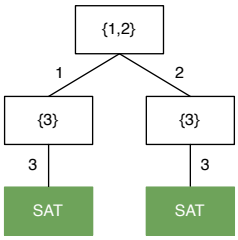


Figure 8.4 – HS-DAG

Figure 8.4 shows an HS-DAG for the above example. Suppose the constraint solver initially returns the unsatisfiable core $\{1, 2\}$, and a root node is created for this core. Then we build an arc for each constraint in the core. In this case, we build two arcs 1 and 2. The left arc is 1, so we remove constraint [1] from the set, and invoke the constraint solver again. This time the constraint solver returns $\{3\}$. We remove constraint [3] and now the constraint set is satisfiable. We create a node SAT for the edge. Similarly, we repeat the same steps for all other edges until all paths reach SAT. Finally, each path from the root to the leaf is a diagnosis. In this case, we have $\{1, 3\}$ and $\{2, 3\}$.

This process alone cannot ensure that the generated diagnoses are minimal. To ensure it, three additional rules are applied to the algorithm. The details of these rules can be found in [GSW89]. Greiner *et al.* [GSW89] prove that HS-DAG builds a complete set of minimal diagnosis after applying the three rules.

8.4.2 From variable sets to fixes

Equipped with the minimal variable sets, we can substitute the configuration values of the variables that do not belong to these sets into c (Step (ii)). Step (iii) is to divide this modified constraint into smaller fix units.

Since the operators in the constraints differ from one language to the other, this task is essentially domain-specific. Nevertheless, since we assume the constraint language is based on quantifier-free predicate logic, we can do some general processing. The basic idea is to convert the constraint into conjunctive

normal form (CNF), and convert each clause into a fix unit. Yet, we still need to carefully make sure the fix units are disjoint and are as simple as possible.

First, if the constraints contain any operators convertible to propositional operators, we convert them into propositional operators. For example, eCos constraints contain the conditional operator ":?" such as $(m\,?\,a : b) > 10$. We convert it into propositional operators: $(\neg m \vee a > 10) \wedge (m \vee b > 10)$.

Secondly, we convert the constraint into CNF. In our example, with $\{m, b\}$, we have $(m \to 6 > 10) \wedge (\neg m \to b > 10) \wedge (6 < b)$, which gives three clauses in CNF: $\{\neg m \vee 6 > 10,\ m \vee b > 10,\ 6 < b\}$.

Thirdly, we apply the following rules repetitively until we reach a fixed point.

Rule 1 Apply constant folding to all clauses.

Rule 2 If a clause contains only one literal, delete the negation of this literal from all other clauses.

Rule 3 If a clause C_1 contains all literals in C_2, delete C_1.

Rule 4 If a clause taking the form of $v = c$ where v is a variable and c is a constant, replace all occurrences of v with c.

In our example, applying Rule 1 to the above CNF, we get $\{\neg m,\ m \vee b > 10,\ 6 < b\}$. Apply Rule 2 to the above CNF, we get $\{\neg m,\ b > 10,\ 6 < b\}$. No further rule can be applied to this CNF.

Fourthly, two clauses are merged into one if they share variables. In the example, we have $\{\neg m,\ b > 10 \wedge 6 < b\}$.

Fifthly, we apply any domain specific rules to simplify the constraints in each clause, or divide the clause into smaller, disjoint ones. These rules are designed according to the types of operators used in the constraint language. In our current implementations of CDL expressions, we use two types of rules. First, for clauses containing only linear equations or inequalities with one variable, we solve them and merge the result. Secondly, we eliminate some obviously eliminable operators, such as replacing $a + 0$ with a. We also apply Rule 1 and Rule 4 shown above during the process. In the example, the second clause consists of two linear inequalities, we solve the inequalities and merge the ranges on b, we get $\{\neg m, b > 10\}$.

Finally, we convert each clause into a fix unit. If the clause has the form of v, $\neg v$, or $v = c$, we convert it into an assignment unit, otherwise we convert it into a range unit. In the example, we convert $\neg m$ into an assignment unit and $b > 10$ into a range unit and get $[m := \texttt{false},\ b :\ b > 10]$.

As mentioned before, the above algorithm does not guarantee the fix units are minimal. The reason is that we cannot ensure that the domain-specific rules in the fifth step are complete, since some common operators such as those on strings are undecidable in general [BTV09].

8.5 Constraint Interaction

So far we have only considered range fixes for one constraint. However, the constraints in variability models are often interrelated; satisfying one constraint might violate another. As a result, we have to consider *multi-constraint* violation rather than single-constraint violation.

Definition 8.2 Multi-constraint violation

A multi-constraint violation *is a tuple* (V, e, c, C)*, where* V *and* e *are unchanged,* c *is the currently violated constraint, and* C *is the set of constraints defined in the model and satisfied by* e*.* ■

The following example shows how a fix satisfying c can conflict with other constraints in C that were previously satisfied.

$$\begin{array}{lcl} V & : & \{m : \texttt{Bool}, n : \texttt{Bool}, x : \texttt{Bool}, y : \texttt{Bool}, z : \texttt{Bool}\} \\ e & : & \{m = \texttt{true}, n = \texttt{false}, x = \texttt{false}, \\ & & \quad y = \texttt{false}, z = \texttt{false}\} \\ c & : & m \wedge n \\ C & : & \{c_2, c_3\} \text{ where} \\ & & \quad c_2 : n \rightarrow (x \vee y) \\ & & \quad c_3 : x \rightarrow z \end{array} \tag{8.2}$$

If we generate a fix from (V, e, c), we obtain $r = [n := \texttt{true}]$. However, applying this fix will violate c_2.

Existing work has proposed three different strategies to deal with this problem; each has its own advantages and disadvantages. We now revisit these three strategies, and show that they can all be used with range fix generation by converting a multi-constraint violation into a single-constraint one. In the evaluation section (Section 8.7) we will give a comparison of the three strategies.

Ignorance

All constraints in C are simply ignored, and only fixes for (V, e, c) are generated. This strategy is used in fix generation approaches considering only one constraint [NEF03]. This strategy does not solve the constraint interaction problem at all. However, it has its merits: first, the fixes are only related to the violated constraint, which makes it easier for the user to comprehend the relation between the fixes and the constraints; secondly, this strategy does not suffer from the problems of incomplete fix list and large fix list, unlike the two others; thirdly, this strategy requires the least computation effort and is the easiest to implement.

Elimination

When a fix violates other satisfied constraints, it is excluded from the list of fixes, i.e., the fix is "eliminated" by other constraints. In the example in violation (8.2), fix r will violate c_2 and thus is excluded from the generated fix set. This strategy is proposed by Egyed *et al.* [ELF08] and used in their UML fix generation tool.

To apply this strategy to range fix generation, we first find a subset of C that shares variables with c, then replace the variables not in c with their current values in e, and connect all constraints by conjunctions. For example, to apply the elimination strategy to violation (8.2), we first find the constraints sharing variables with c, which includes only $c2$, and then replace x and y in c_2 with their current values, getting $c_2' = n \rightarrow \texttt{false} \vee \texttt{false}$. Then we generate fixes for $(V, e, c \wedge c_2')$.

Although the elimination strategy prevents the violation of new constraints, it has two noticeable drawbacks. First, it excludes many potentially useful fixes. In many cases, it is inevitable to bring new errors during error resolution. Simply excluding fixes will only provide less help to the user. In our example, we will get an empty fix set, which does not help the user resolve the error. Secondly, since we need to deal with the conjunction of several constraints, the resulting constraint is much more complex than the original one. Our evaluation showed that some conjunctions can count more than ten constraints. Nevertheless, compared to the propagation strategy, this increase in complexity is still small.

Propagation

When a fix violates other constraints, we further modify variables in the violated constraints to keep these constraints satisfied. In this case, the fix is "propagated" through other constraints. For example, fix r will violate c_2, so we further modify variables x or y to satisfy c_2. Then the modification of x will violate c_3, and we further modify z. In the end, we get two fixes $[n := \texttt{true}, x := \texttt{true}, z := \texttt{true}]$ and $[n := \texttt{true}, y := \texttt{true}]$. This approach is used in the eCos configuration tool [VD01], and the feature model diagnosis approach proposed by White *et al.* [WSB+08].

To apply this strategy, we first perform a static slicing on C to get a set of constraints directly or indirectly related to c. More concretely, we start from a set D containing only c. If a constraint c' shares any variable with any constraint in D, we add c' to D. We keep adding constraints until we reach a fixed point. Then we make a conjunction of all constraints in D, and generate fixes for the conjunction. For example, if we want to apply the propagation strategy to violation (8.2), we start with $D = \{c_1\}$, then we add c_2 because it

shares n with c_1, next we add c_3 because it shares x with c_2. Now we reach a fixed point. Finally, we generate fixes for $(V, e, c_1 \wedge c_2 \wedge c_3)$.

The propagation strategy ensures that no satisfied constraint is violated, and no fix is eliminated. However, there are two new problems. First, the performance cost is the highest among the three strategies. The constraints in real-world models are highly interrelated. In large models, the strategy often led to conjunctions of hundreds of constraints. The complexity of analyzing such large conjunctions is significantly higher than analyzing a single constraint. Secondly, since many constraints are considered together, this strategy potentially leads to large fixes (i.e., fixes that modify a large set of variables), and large number of fixes, which are not easy to read and to apply.

8.6 Implementation

We have implemented a command-line tool generating fixes for eCos CDL using the Microsoft Z3 SMT solver [DMB08]. Our tool takes a CDL configuration as input, and automatically generates fixes for each configuration error found. Alternatively, the user can enter an option to activate via the command-line interface, and our tool generates fixes to activate this option.

To implement our algorithm, one important step is to convert the constraint in the CDL model into the standard input format of the SMT solver: SMT-LIB [BST10]. To perform this task, we carefully studied the formal semantics of CDL [Xio11, BS10a] through reverse engineering from the configurators and the documents. We faced two problems during the conversion. First, CDL is an untyped language, while SMT-LIB is a typed language. To convert CDL, we implement a type inference algorithm to infer the types of the options based on their uses. When a unique type cannot be inferred or type conflicts occur, we manually decide the feature types.

The second problem is dealing with string constraints. The satisfiability problem of string constraints is undecidable in general [BTV09], and general SMT solvers do not support string constraints [DMB08]. Yet, string constraints are heavily used in CDL models. Nevertheless, our previous study on CDL constraints [PNX+11] actually shows that the string constraints used in real world models employ a set semantics: a string is considered as a set of substrings separated by spaces, and string functions are actually set operations. For example, `is_substr` is actually a set member test. Based on this discovery, we encode each string as a bit vector, where each bit indicates whether a particular substring is present or not. Since in fix generation we will never need to introduce new substrings, the size of the bit vector is always finite and can be determined by collecting all substrings in the model and the current configuration.

Table 8.1 – Real World Configuration Files

Architecture	Project	Options	Constraints	Changes
virtex4	ReconOS	933	330	49
xilinx	ReconOS	765	272	53
ea2468	redboot4lpc	658	96	14
aki3068net	Talktic	817	195	3
gps4020	PSAS	535	85	23
arcom-viper	libcyt	771	189	26

8.7 Evaluation

8.7.1 Methodology

Our algorithm ensures Properties 1-4 for the generated range fixes. However, to really know whether the approach works in practice, several research questions need to be answered by empirical evaluation:

- **RQ8.1** *How complex are the generated fix lists?*
- **RQ8.2** *How often are the final user changes covered by our fixes?*
- **RQ8.3** *How efficient is our algorithm?*
- **RQ8.4** *Does our approach cover more user changes than existing approaches?*
- **RQ8.5** *What are the differences among the three strategies?*

The evaluation uses 6 eCos configuration files from 5 eCos-based open-source projects (Table 8.1). Each file targets a different hardware architecture (the first column in Table 8.1); each architecture uses a different mixture of eCos packages, yielding variability models with different options and constraints (columns three and four). The configuration process for a given model starts from the model's default configuration; the last column in Table 8.1 specifies the number of changes made by a project to a default configuration.

The evaluation needs a set of real-world constraint violations. Interestingly, the default configuration for each model already contains *errors*—violations of requires constraints. The first column in Table 8.2 shows their numbers. The models share common core packages, causing duplicated errors. A set of 68 errors from defaults remain after removing duplicates.

For RQ2 and RQ4, we attempt to recover the sequence of user changes from the revision history of the configuration files. We assume that the user starts from the default configuration and solves errors from defaults by accepting

Table 8.2 – Constraint violations

Architecture	Errors in defaults	Errors in changes	Activating
virtex4	56	5	15
xilinx	48	1	2
ea2468	8	8	1
aki3068net	26	3	0
gps4020	12	10	4
arcom-viper	26	0	0

the suggestions from the eCos configurator. We record this corrected default configuration as the first version. Then we diff each pair of consecutive revisions to find changes to options. Next we replay these changes to simulate the real configuration process. Since we do not know the order of changes within a revision, we use three orders: a top-down exploration of the configuration file, a bottom-up one, and a random one. The rationale for the first two orders is that expert users usually edit the textual configuration file directly rather than using the graphical configurator. In this case, they will read the options in the order that they appear in the file, or the inverse if they scroll from bottom to top.

We replay the changes as just explained and collect (i) *errors*—violating requires constraints—and (ii) *activation violations*. An activation violation occurs when an option value should be changed, but is currently inactive. The last two columns in Table 8.2 show the numbers of the resulting violations from changes. After duplicate removal, 27 errors and 22 activation violations remain; together with the first dataset, we have a total of 117 multi-constraint violations.

Finally, we invoke our tool to generate fixes for the 117 violations. For RQ4, we also invoke the built-in fix generator of the eCos configurator on the 27 errors from the user changes. The activation violations are not compared because they are not supported by the eCos configurator. The experiments were executed on a computer with an Intel Core i5 2.4 GHz CPU and 4 GB memory.

8.7.2 Results

We first give the results for RQ8.1-RQ8.4 using the propagation strategy. We answer RQ8.5 by presenting the comparison of the three strategies last.

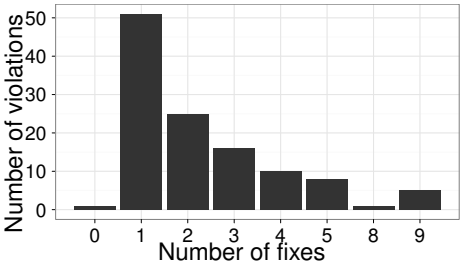


Figure 8.5 – The number of fixes per violation

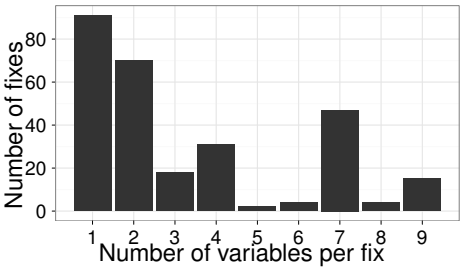


Figure 8.6 – The number of variables per fix

RQ8.1

To answer RQ8.1, we first consider two basic measures over the 117 violations: the distribution of the number of fixes per violation (see Figure 8.5), and the distribution of the number of variables changed by each fix (see Figure 8.6). From these figures we see that most fix lists are short and most fixes change a small number of variables. More concretely, 95% of the fix lists contain at most five fixes, and 75% of the fixes change less than five variables. There is also an activation violation that did not produce any fix. A deeper investigation of this violation revealed that the option is not supported by the current hardware architecture, and cannot be activated without introducing new configuration errors. The extracted changes actually lead to an unsolved configuration error in the subsequent version.

It is still unclear how the combination of fix number and fix size affect the size of a fix list, and how the large fixes and long lists are distributed in the violations. To understand this, we measure the size of a fix list. The size of a fix list is defined as the sum of the number of variables in each fix. The result is shown in Figure 8.7. From the figure we can see that the propagation strategy does lead to large fix lists. The largest involves 58 variables, which is

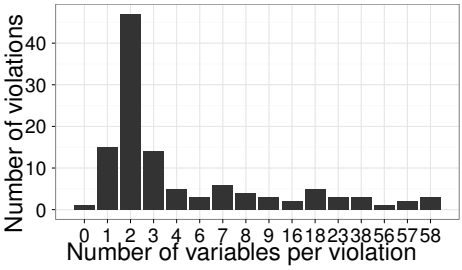


Figure 8.7 – The sizes of fix lists

not easily readable. However, the long lists and large fixes tend to appear only on a relatively few number of violations, and the majority of the fix lists are still small: 83% of the violations contains less than 10 variables.

We also measure the number of variables in each fix unit to understand how large the fix units are. It turns out that every fix unit contains only one variable. This shows that (1) "minimality of fix units" effectively holds on all the violations, and (2) ranges declared on more than one variable (such as the second fix for violation (8.1)) have never appeared in the evaluation.

RQ8.2

Given an error or activation violation, we examined the change history to identify a subsequent configuration that corrected the problem. To answer RQ8.2, we checked if the values in the corrected configuration fell within one of the ranges proposed by our generated fixes.

There are in total 47 out of 49 violations with subsequent corrections in our dataset. The fixes generated by our tool covered 46 of these violations (98%). An investigation into the remaining violations showed that the erroneous option discussed in RQ8.1 is responsible for that discrepancy. Since the propagation strategy ensures no new error is introduced, the resolved value from the dataset was not proposed as a fix.

RQ8.3

For each of the 117 violations, we invoked the fix generator 100 times, and calculated the average time. The result is presented as a density graph in Figure 8.8. It shows that most fixes are generated within 100 ms. Some fixes require about 200 ms, which is still acceptable for interactive tools.

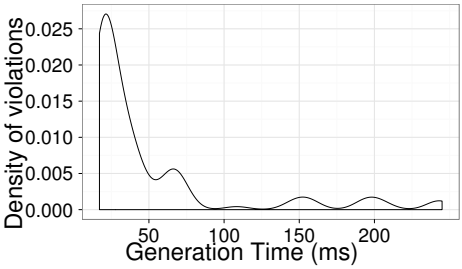


Figure 8.8 – Fix generation time

RQ8.4

We measure whether the fixes proposed by the eCos configurator cover the user changes in the same way as in RQ8.2. Since the eCos configurator is unable to handle the activation violations, we measure only error resolutions. There are 26 out of 27 errors that have subsequent corrections. The eCos configurator was able to handle 19 of the 26 errors, giving a coverage of 73%. Comparatively, our tool covered all 26 errors.

RQ8.5

As discussed in Section 8.5, the propagation strategy potentially produces large fix lists. At this stage, we would like to know if the other two strategies actually produce simpler fixes. We compare the size of fix lists generated by the three strategies in Figure 8.9. The elimination and ignorance strategies completely avoid large fix lists, with the largest fix list containing four variables in total. The elimination strategy changes even fewer variables because some of the larger fixes are eliminated.

We also compare the generation time of the three strategies. For all violations, the average generation time for the propagation strategy is 50ms, while the elimination strategy is 20ms and the ignorance strategy is 17ms. Since the overall generation time is small, it does not make a big difference in tooling.

Next, we want to understand to what extent the other two strategies affect completeness or bring new errors. First we see that the elimination strategy does not generate fixes for 17 violations. This is significantly more than the ignorance and propagation strategies, which have zero and one violation, respectively. We measure the coverage of user changes using the elimination strategy. In the 47 violations, only 27 are covered, giving a coverage of 57%. This is even lower than the eCos configurator, which generates only one fix, showing that a lot of useful fixes were eliminated by this strategy.

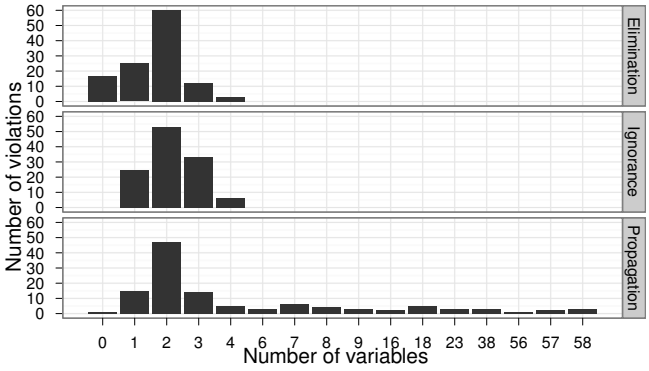


Figure 8.9 – The sizes of fix lists in the three strategies

The problem of the ignorance strategy is that it may bring new errors. To see how frequently a fix brings new errors, we compare the fix list of the ignorance strategy with the fix list of the elimination strategy. If a fix does not appear in the list of the elimination strategy, it may potentially bring new errors. As a result, 32% of the fixes generated by the ignorance strategy bring new errors, which covers 44% of the constraint violations. This shows that the constraints in practice are usually inter-related and the ignorance strategy potentially causes new errors in many cases.

8.8 Threats to Validity

We see two main threats to external validity. First, we have evaluated our approach on one variability language. However, Berger *et al.* [BSL+10] study and compare three variability languages—CDL, Kconfig and feature modeling—and find that CDL has the most complex constructs for declaring constraints, and constraints in CDL models are significantly more complex than those in Kconfig models. Thus, our result is probably generalizable to the other two other languages.

The second threat is that our evaluation is a simulation rather than an actual configuration process. We address this threat by using the models of six architectures and configurations gathered from five projects. The configurations and changes have a wide range of characteristics as shown in Tables 8.1 and 8.2. However, it still may be that these changes are not representative of the problems that real users encountered. We hope to address this threat by running a user study in industry settings in the future.

A threat to internal validity is that our translation of CDL into logic con-

straints could be incorrect. To address this threat, we have developed a formal specification of CDL semantics in functional style, in addition to the one developed by Berger *et al.* [BS10a]. We have carefully inspected and compared both against each other and tested them on examples with respect to the eCos configurator.

8.9 Related Work

The idea of automatic fix generation is not new. Nentwich *et al.* [NEF03] propose an approach that generates abstract fixes from first-order logic rules. Their fixes are abstract because they only specify the variables to change and trust the user to chose a correct value. In contrast, our approach also gives the range of values for a variable. Furthermore, their approach only supports "$=$" and "$\neq$" as predicates and, thereby, cannot handle models like eCos. Scheffczyk *et al.* [SRBS04] enhance Nentwich *et al.* 's approach by generating concrete fixes. However, this approach requires manually writing fix generation procedures for each predicate used in each constraint, which is not suitable for variability models, often containing hundreds of constraints. Egyed *et al.* [ELF08] propose to write such procedures for each type of variable rather than each constraint to reduce the amount of code written and apply this idea to UML fix generation. Yet, in variability models, the number of variables is often larger than the number of constraints. The actual reduction of code is thus not clear. Jose *et al.* [JM11] generate fixes for programming bugs. They first identify the potentially flawed statements using MAXSAT analysis, and then propose fixes based on heuristic rules. However, their heuristic rules are specific to programming languages and are not easily applicable to software configuration. Also, they propose at most one fix each time rather than a complete list.

Fix generation approaches for variability models also exist. The eCos configurator [VD01] has an internal fix generator, producing fixes for a selected error or on-the-fly when the user changes the configuration. White *et al.* [WSB+08] propose an approach to generate fixes that resolve all errors in one step. However, both approaches can only produce one fix rather than a complete list. Furthermore, they have very limited support of non-Boolean constraints. White *et al.* 's approach does not handle non-Boolean constraints at all, while the eCos configurator supports only non-Boolean constraints in a simple form: $v \oplus c$ where v is a variable, c is a constant, and $\oplus$ is an equality or inequality operator.

Another set of approaches maintain the consistency of a configuration. Valid domains computation [HSJ+04, Men09] is an approach that propagates decisions automatically. Initially all options are set to an unknown state. When the user assigns a value to an option, it is recorded as a decision, and all other options whose values are determined by this decision are automatically set.

In this way, no error can be introduced. Janota *et al.* [JBGMS10] propose an approach to complete a partial configuration by automatically setting the unknown options in a safe way. However, both approaches require that the configuration starts with variables in the unknown state. Software configuration in practice is often "reconfiguration" [BSL+10], i.e., the user starts with a default configuration, and then makes changes to it. In reconfiguration cases, variables are assigned concrete values rather than the unknown state. Furthermore, these approaches are designed for small finite domains, and it is not clear whether they are scalable to large domains such as integers.

Several approaches have been proposed to test and debug the construction of variability models themselves. Trinidad *et al.* [TBD+08] use Reiter's theory of diagnosis [Rei87] to detect several types of deficiencies in FODA feature models. Wang *et al.* [WXH+10] automatically fix deficiencies based on the priority assigned to constraints. These approaches target the construction of variability models and cannot be easily migrated to configuration.

Others automatically fix errors without user intervention. Demsky and Rinard [DR03] propose an approach to fix runtime data structure errors according to the constraint on the data structure. Mani *et al.* [MSDS10] use the hidden constraints in a transformation program to fix input model faults. Xiong *et al.* [XHZ+09] propose a language to construct an error-fixing program consistently and concisely. Compared to our approach, these approaches also infer fixes from constraints, but they only need to generate one fix that is automatically applied. Completeness is not considered by these approaches.

The HS-DAG algorithm is often used in combination with the QuickXPlain algorithm [Jun04]. The QuickXPlain algorithm computes the preferred explanations and relaxations for over-constrained problems. This combination has been successfully applied in recommender systems to find the most representative relaxations of a set of requirements, i.e., those with highest likelihood of being chosen by the users [FFJS04, JL06, FFS+09]. O'Sullivan *et al.* [OPFP07] propose an alternative algorithm for the same problem. The most representative relaxations are then used to propose alternative solutions based on a database of known operational solutions. The filtering of fixes is a possible extension to our work.

8.10 Towards Collaborative Conflict Resolution

The fix generation algorithm lays the foundation stones for both individual and collaborative conflict resolution. Until now, we implicitly assumed a single-user mode in which conflicts are fixed as soon as they appear. This is in fact the most frequent scenario played by Linux and eCos users [HXC12]. However, in collaborative environments, such as multi-view FBC, conflict resolution involves several users. Several users means different viewpoints to reconcile, priorities

to respect, security policies to enforce (e.g., visibility and access rights), and workflows to follow.

The collaborative context does not influence the execution of the fix generation algorithm; assuming the FM is shared by all the users. To reason about the decisions from n users, the current configuration is determined by the union of their decisions, i.e., $e = e_1 \cup \ldots \cup e_n$. Unlike the single-user case, e could contain multiple assignments for the same variable. For instance, the union of the two configurations $e_1 : \{m = \texttt{true}, a = 12, b = 12\}$ and $e_2 : \{m = \texttt{true}, a = 5, b = 6\}$ contains different assignments for a and b, which produce different fixes.

To tackle this problem, the inconsistent assignments can be corrected either prior to generating fixes or during fix generation. Solving inconsistencies prior to fix generation basically means presenting the users with a list of conflictual assignments and asking them to agree on a common value. This approach does not require any reasoning from the tool. The flip side of the medal is that changing the values can introduce several new conflicts. Conversely, when conflicting assignments are solved during conflict resolution, the generated fixes will guarantee that no new conflict is introduced. This, however, comes at the cost of potentially large and numerous fixes because all the combinations of conflictual assignments have to be explored. The first challenge is to determine which approach is most appropriate.

How and when a conflict is fixed is also an open question. Different merge operations (e.g., three way merging and octopus merging) have been successfully implemented in alternative SCM tools (e.g., Subversion [Apa11] and Git [Git11]) and models (e.g., *checkout/checkin*, *composition*, *long transaction*, and *change set* [Fei91]). Collaborative software development platforms like IBM Jazz [IBM11] target distributed teams, and aim to achieve better productivity and transparent result integration. Such tools and models establish communication channels between users, and regulate their interactions. They are the breeding ground for collaborative fix generation. The second challenge is to accommodate the organizational constraints they impose with the input and output of the fix generation algorithm.

8.11 Chapter Summary

Range fixes provide alternative solutions to constraint violations in software configuration. They are correct, minimal in the number of variables per fix, maximal in their ranges, and complete. We also evaluated three different strategies for handling the interaction of constraints: ignorance, elimination, and propagation. The evaluation with eCos showed that the propagation strategy provides the most complete fix lists without introducing new errors, and the fix sizes and generation times are within acceptable ranges. However, if

more complex situations are encountered, elimination or ignorance can provide simpler fix lists and faster generation time, at the expense of completeness or the guarantee not to introduce new errors. We finally discussed the challenges ahead to integrate fix generation in collaborative software development.

Chapter

9 A Toolset for Advanced Feature-based Configuration

9.1 Overview

Chapters 5 to 8 laid the foundations for multi-view FMs, FCWs, and conflict detection and resolution. We now present a toolset for FCW[1] that has been implemented by extending and integrating two third-party tools: SPLOT [Men10][2] and YAWL[3].

FM management with SPLOT. SPLOT is an open source web-based system for editing, sharing, and configuring FMs. The public version of SPLOT available online now gathers 100+ FMs that are all freely accessible. SPLOT is developed in Java, and uses Freemarker[4] and Dojo[5] to handle the web front-end. To provide efficient interactive configuration, SPLOT relies on a SAT solver (SAT4J[6]) and a BDD solver (JavaBDD[7]). Their reasoning abilities enable decision propagation and critical debugging operations [Men09].

SPLOT was chosen because it provides a simple, yet robust, web-based visual editor to create, edit, and configure FMs. Its servlet-based architecture makes it easy to extend. Third-party software can interact with SPLOT through web-services. The existing repository of FMs is also an excellent testbed for our extensions. We extended it to support view creation, configuration, and view-to-workflow mapping.

[1] http://www.splot-research.org/extensions/fundp/fundp.html
[2] http://www.splot-research.org/
[3] http://www.yawlfoundation.org/
[4] http://freemarker.sourceforge.net/
[5] http://www.dojotoolkit.org/
[6] http://www.sat4j.org/
[7] http://javabdd.sourceforge.net/

Workflow management with YAWL. The YAWL tool, named after the language, is a full-fledged workflow management environment that enables workflow modelling and analysis, and provides web service integration. The workflow modelling tool is a standalone application implemented in Java while the runtime environment is web-based.

The YAWL tool is developed based on a service-oriented architecture (SOA). Third-party extensions are provided as web services that can add extra controls and functionalities to the base system. The mapping to YAWL from other languages such as BPMN, BPEL, and activity diagrams [WVvdA+09] adds an extra level of genericity to our approach. Interactive services were added to YAWL to start view-based configuration in SPLOT.

FCW Engine The cornerstone of this new configuration environment is the *FCW Engine*, which we implemented from scratch. Its role is to manage configuration sessions, convey the information between YAWL and SPLOT, and monitor the whole configuration process.

9.2 Integrated Toolset

Figure 9.1 shows the essential components of our integrated toolset as well as a typical usage scenario. All the elements under YAWL and SPLOT are web service extensions. The newly developed FCW Engine minimizes the coupling between YAWL and SPLOT. For clarity, we discuss the design and runtime phases of the scenario separately. Then, we elaborate on user management and explain how concurrent configuration is achieved. Design time activities being independent of the configuration process, they can be performed either during domain or application engineering [PBvdL05]. In other words, SPL engineers are free to define a generic FCW during domain engineering, or they can tailor an FCW for every application. To obtain more flexibility, *configurable* workflows [GvdAJVLR07] could also be used during domain engineering, and then configured during application engineering.

9.2.1 Design time: Configuration preparation

View specification in SPLOT. The specification of views is a parallel activity to the design of the workflow (❶). The *Design views* service is a new extension to SPLOT that provides the user with a web form to specify and edit FM views. A view is defined intensionally with an XPath-like expression (see Section 5.3.2). Basically, the XPath expression specifies paths to features in the FM that should be part of the view. A coverage test can be run to verify that the whole FM can be configured through the defined views, i.e., that no feature can be left undecided after the views have been configured (see Section 5.3.3). The views are then saved in an XML repository.

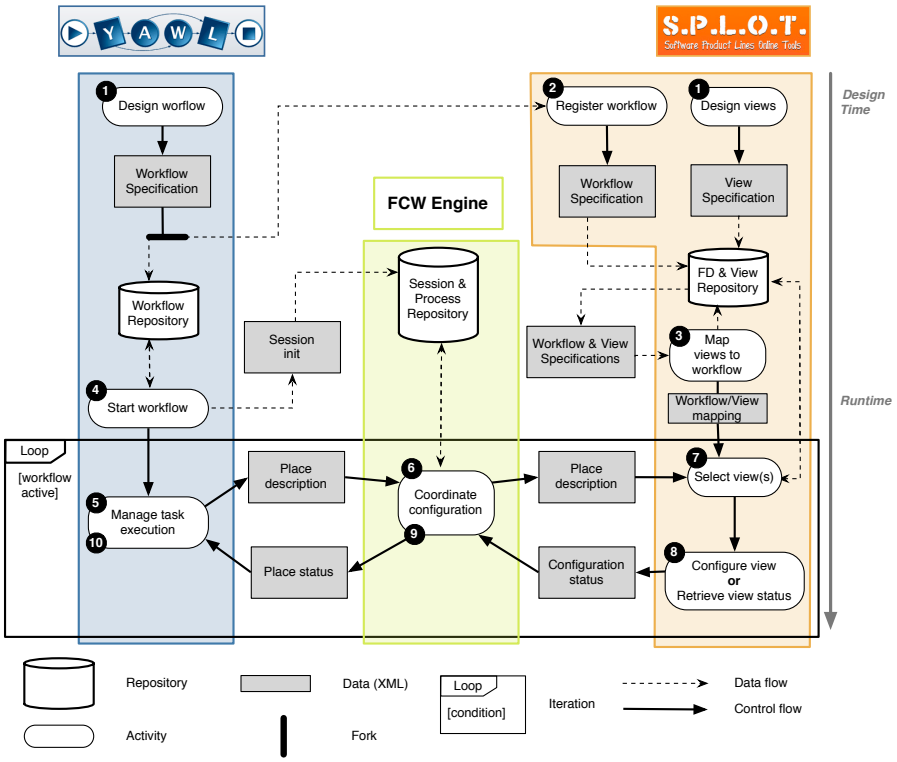


Figure 9.1 – Overview of the essential components and typical use case scenario.

Workflow specification in YAWL and registration. The YAWL workflow editor allows to design the workflow and store it in the repository (❶). Some fundamental properties of the workflow can also be verified like (weak) soundness (see Section 7.6.1). Internally, we link a YAWL custom service to each task and condition. This service is responsible for monitoring the activation and completion of the associated task or condition. Once created and checked, the workflow specification can be uploaded and registered in SPLOT (❷). During the registration process, the workflow is parsed and the information necessary for the mapping with the views is extracted.

View-to-task mapping in SPLOT. The mapping of views to tasks is a key activity as it determines when a view is going to be configured. We have seen in Section 7.4 that a view not only has to be mapped to a task that triggers its configuration, but also to a *stop* that tells when it should be fully configured. The stop is materialized in the workflow by a condition. The mapping of a view to its task and stop is performed in SPLOT (❸). The mapping is correct and

complete when (1) all the views are mapped to exactly one task and one stop, and (2) the coverage of the mapped views is complete.

9.2.2 Runtime: Product configuration

Workflow initialization in YAWL. The FCW Engine is a Java-based system that drives the configuration process. Its first duty is to manage multiple configuration sessions that can be started from YAWL (❹). As the workflow is executed, the tasks and stops are activated, which calls the associated web services (❺).

Configuration management with the FCW Engine.

The second duty of the FCW Engine is to control the progress of a configuration session. The FCW Engine provides a control panel that allows to monitor the status of tasks and conditions. The status of a task in a given instance can either be *Ready*, *Configured*, or *Completed*. Similarly, the status of the stops can either be *Ready* or *Completed*. The *Coordinate configuration* service handles messages received from YAWL and SPLOT. When an element is activated in YAWL, the web service sends its name, its type (task or condition), and the session information to the FCW Engine (❻).

View-based Configuration in SPLOT. The FCW Engine initiates either a view configuration request if the element is a task, or a configuration status request if it is a condition. If it is a configuration request, SPLOT loads the corresponding view (❼). The user then has to choose one of the three visualizations (see Section 5.3.4).

In the interactive configuration form, the user performs the configuration by selecting/deselecting the features (❽). The existing services of SPLOT control the configuration process to guarantee that only valid decisions are made. SPLOT has been extended to support partial and complete configuration, persistency and recovery, and decision logging.

When the configuration of the view is terminated, the FCW Engine updates the status of the task (❾), and the user can mark the task as complete in YAWL (❿).

If the place is a condition, the FCW Engine requests the list of views attached to the stop (❼). SPLOT returns the status of each of the views to the FCW Engine (❽), which then checks whether the stop is satisfied, i.e., whether all the views are completely configured (❾).

When the output condition is reached, the configuration stops, and the final condition is checked. The resulting product can be retrieved from the repository in SPLOT.

9.3 User Management

The mapping of a view to a task and a condition is only a technical step. The actual assignment of a user to the configuration of a view is done in YAWL. The toolset uses YAWL's user management functionality [Ada10]. YAWL allows to create users, and to customise several parameters such as roles, capabilities, positions, groups and privileges. In the Spacebel case, for example, five users are created: Spacebel, the reseller, the system engineer, the TMTC integrator, and the network integrator.

At runtime, when a task is ready to be executed, the administrator of the running instance of the FCW either assigns the task directly to a user or to a role, meaning that all the users with the same role can execute it. When a user logs in, he can only access and start the active tasks that have been assigned to him. The user ID is part of the data that is communicated to the FCW Engine and then relayed to SPLOT. This way, both the FCW Engine and SPLOT can keep track of who made what decision and when.

9.4 Multi-view Feature Modelling

This section dwells upon the three core extensions for multi-view FBC provided by the toolset.

The first extension enables view creation with XPath expressions. Figure 9.2 shows the view creation menu of SPLOT. The upper part shows the FM of PloneMeeting. In the middle part, views can be created or edited. Here, the User view is selected and the XPath expression that defines it is displayed. The bottom part contains additional information identifying the creator of the view. Finally, the *Evaluate XPath Expression* button checks that the XPath expression is correct and shows the results of its evaluation. The *Evaluate Views Coverage* button checks the completeness of the views and returns the features that are not covered, if any. The last two buttons save, respectively delete, the current view in the shared repository.

The actual configuration of a view is provided by the second extension. The extension allows to select (1) the view to configure, and (2) the visualisation. In Figure 9.3, the view of the User is selected and the pruned visualisation is activated. Note the greyed *Data* feature: it can neither be selected nor deselected. The stakeholder can switch freely from one visualisation to another as she configures her view without loosing the decisions that were already made. This way, we *dynamically combine* the advantages of the three visualisations and leave the complete freedom to the stakeholder to choose the one(s) that best fit(s) her preferences.

The table on the right monitors the status of the current configuration. Basically, it tells what features have been selected or deselected, and which

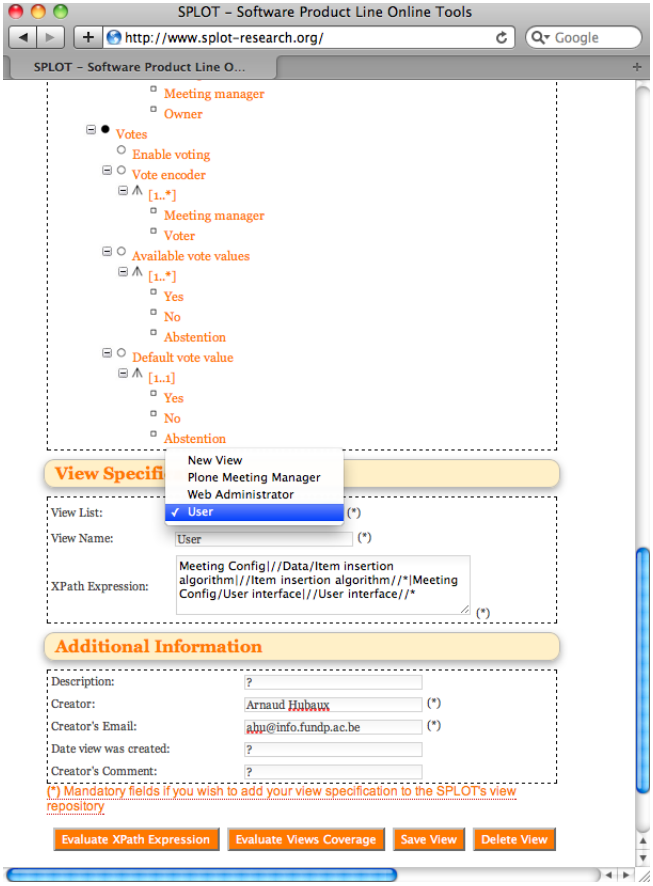


Figure 9.2 – SPLOT view creation menu illustrated on the User view.

decisions were propagated. It also provides general information about the operations performed by the SAT solver and the status of the configuration. The latter is a good indicator of the work that remains after the configuration of the view. As we have seen, the solver reasons about the full FM and not only about individual views. This is important in practice. Recall that for the collapsed visualisation, Section 5.5.2 concludes that the cardinalities produce an under-constrained FM. Cardinalities are part of the constraints taken into account by the solver. Thereby, the decision to select or deselect a feature in the view is propagated in the complete model—keeping the global configuration consistent. In the counter-example in Figure 5.6 for instance, the selection of d in the view will automatically entail the selection of c, even though the

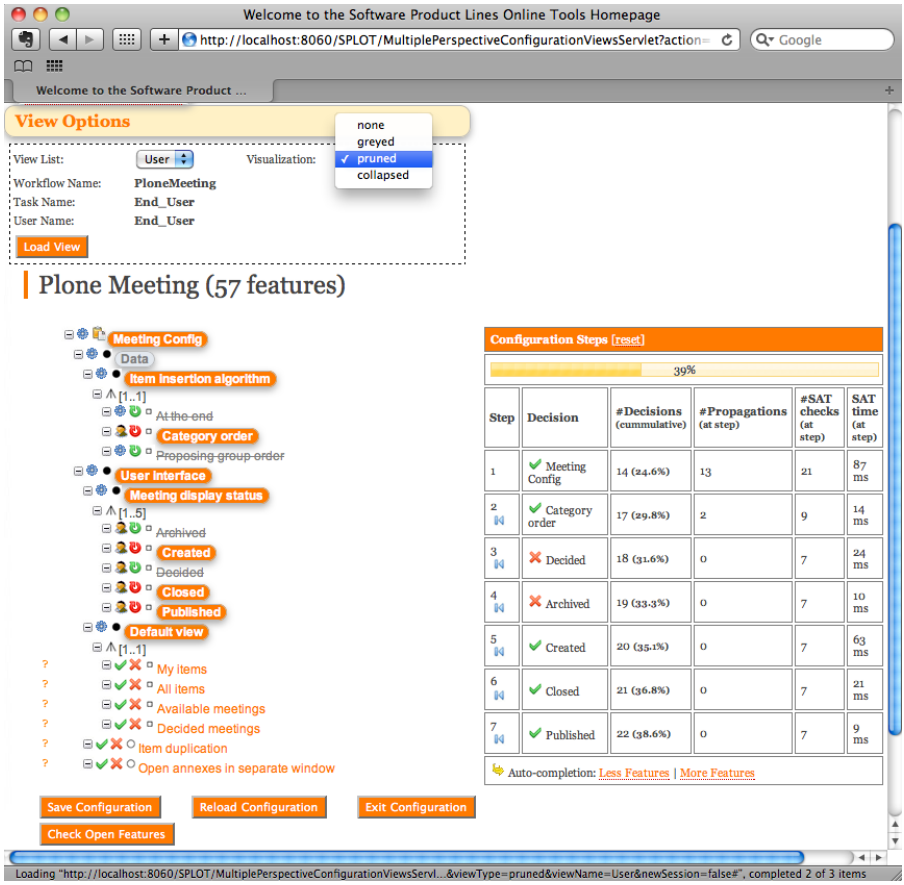


Figure 9.3 – Configuration view of the User with the pruned visualisation in SPLOT.

recomputed cardinality does not enforce that propagation.

The third extension provides basic support for multi-user concurrent configuration. At the time being, it only enables synchronous configuration. To prevent conflictual decisions, a configuration session manager is used (see Section 9.6). Its role is (1) to maintain a mutual exclusion on the configuration engine so that only one user can commit a decision at a time, and (2) to notify all the users of a decision and of the results of the propagation.

9.5 Verification and Execution

Design time verifications (Section 7.7) are performed after the mapping of views onto the workflows (❸). The mapping is then used to generate the transformed workflow. Each individual workflow has then to be manually loaded in YAWL for analysis. If the workflow requires corrections, it will have to be loaded in SPLOT again and the mappings redefined.

Strong and weak dependency sets (Section 7.6) are also computed after the mapping. They are stored in SPLOT with the view descriptions and the mapping. The dependency sets are loaded by SPLOT when the configuration menu of a given view is activated. The status of open features (Section 7.7) can be obtained at any time during the configuration with the *Check Open Feature* button. Open features are also automatically checked when the user exists the configuration menu. Those that are not propositionally defined by the weak and strong dependency sets are reported to the user.

Originally, a final check on open features was supposed to be performed by the YAWL's workflow management environment when a configuration task is marked as completed. However, restrictions in YAWL's source code did not allow us to override or wrap the task completion method.

9.6 Concurrent Configuration Management

Two levels of concurrency can be achieved with the toolset. An FCW instance defines the first level of concurrency and is basically a complete configuration project. Each FCW instance is given an ID by the *Registration* service of the FCW Engine, which allows to retrieve and load the proper configuration from the repository. Within an FCW instance several configuration sessions can run in parallel. That is the second level of concurrency. A configuration session is the period during which a user executes a configuration task. Each part of the toolset contributes to handle concurrent configuration.

YAWL has a built-in mechanism to run multiple instances of the same workflow, where a particular instance is called a *case*. Users can switch between running cases and proceed with their tasks in each individual case. YAWL thus natively provides support for both levels of concurrency.

All the FCW instances and configuration sessions are monitored by the FCW Engine. To manage concurrent sessions within the same FCW instance, the FCW Engine coordinates users with active configuration tasks and forwards the list of active users to SPLOT. SPLOT allocates distinct configuration spaces for each FCW instance. To keep the configuration space consistent when multiple configuration sessions are simultaneously active, we have implemented a basic conflict avoidance mechanism. In essence, an FCW Instance Manager is created for every FCW instance. It plays two major roles. First, it controls

the access to the reasoning engine that preserves the consistency of the configuration space. Only the manager can submit decisions and receive propagation results from the reasoning engine. Active users submit their decisions to the manager. The manager maintains a lock on the reasoning engine so that only one decision is processed at a time. This prevents conflictual decisions within the same configuration space. Secondly, when a decision has been processed by the reasoning engine, the manager notifies all the active users of the decision and the results of the propagation. Only then does the manager release the lock. That simple mechanism enables conflict-free synchronous configuration of a multi-view FM.

The range fix algorithm still requires an extra layer on top of it to manage user interactions before being integrated into the toolset. At the technical level, the SMT solver used in our prototype (Z3) is a Microsoft product that hardly blends in our open source environment. We are currently investigating the replacement of Z3 by STP[8]. Also, the expressiveness of SPLOT should be extended to include non-Boolean variables to benefit from the complete reasoning power of our algorithm.

9.7 Chapter Summary

This chapter brought together the contributions of Chapters 5–7 into a process-aware and multi-user FBC toolset. Our ambition was to demonstrate the applicability and scalability of FBC tools with a prototype based on SPLOT to support: (1) view editing, rendering and configuration; (2) views-to-workflow mapping; (3) configuration persistency and recovery. Workflow execution is supported through YAWL which was extended to monitor task and condition execution and completion. An FCW Engine was created to control the interactions between SPLOT and YAWL, and manage the configuration sessions. We also implemented the design time and runtime analyses that ensure the proper completion of the configuration process.

Inconsistencies during the concurrent configuration of the FM are prevented by a lock on the configuration space. The underlying assumption here is that views are configured synchronously. We have seen in Chapter 8 that this assumption rarely holds in practice. We are now studying the application of range fixes to concurrent configuration.

[8] See `http://sites.google.com/site/stpfastprover/`

Chapter

10 Conclusion

10.1 Our Vision

American naturalist, philosopher, and writer, Henry David Thoreau (1817–1862) summarised in [Tho54] the vision behind this work:

> *If you have built castles in the air, your work need not be lost; that is where they should be. Now put the foundations under them.*

Pressured by a demand for customisable products, manufacturers and software vendors have adopted sophisticated configurators to manage variations in context and requirements. Configuration options are usually specified with a dedicated language, such as the feature modelling language. The feature model structures and describes the configuration options in a hierarchy of features whose selection is ruled by crosscutting constraints. Over the years, the development of open source and commercial feature-based configuration tools echoed with advances in the formalisation and analyses of FMs.

Promoted by software product lines and advances in tool support, feature-based configuration has been used to tackle increasingly complex configuration problems. That complexity is marked by a higher number of features, constraints, and, most importantly, users. These multiple users add a new dimension to feature-based configuration: *collaboration*.

Early in this research, we ran into the *chicken-and-egg* problem: to create a solution, more detailed requirements were needed, and to obtain detailed requirements, an operational prototype seemed necessary.

Collaborative feature-based configuration is still in the incubation phase. When we started studying it, the practitioners we met were able to describe

high-level challenges but struggled to formulate precise requirements. Essentially, they all expressed a need for enhanced control and dependability but no comprehensive requirements for a solution.

Surprisingly, we often felt that the absence of real support for collaborative feature-based configuration was an obstacle to requirement elicitation. *Ad hoc* solutions (software or not) are usually implemented to patch isolated problems, thereby losing track of the actual problem. Yet, existing support is regularly used as a yardstick to identify what is missing or should be improved. To obtain more definitive requirements, we decided to formally specify and implement a frame of reference for collaborative configuration. This frame of reference provides a foundation upon which domain-specific tools can be built to assess and refine the initial set of requirements. We summarise below how this frame of reference was developed, and outline perspectives.

10.2 Summary of the Contribution

The central argument of this thesis is that collaborative feature-based configuration can be achieved by providing control over the decomposition, assignment, and execution of configuration tasks. Our contribution rests on sound mathematical definitions, properties, and theorems, which have been implemented in an open source toolset. This toolset produces evidence of the applicability, efficiency, and reliability of our definitions and algorithms. More precisely, the four research questions posed in the problem statement (see Section 1.3) have been answered as follows:

RQ1 *How to achieve separation of concerns in FMs?* To grasp a better understanding of how separation of concern is achieved in feature-based configuration, we conducted a systematic literature survey. We notably found separation and composition techniques for concerns in feature models to be rudimentary, which makes realistic feature models hard to comprehend and analyse. To tackle that problem, we studied PloneMeeting (a web-based meeting management software product line), several multi-perspective feature-based configuration techniques, and other configuration tools. From that investigation, we formally defined *multi-view feature models*, a view specification mechanism, necessary and sufficient coverage conditions, and three visualisation alternatives disclosing features on a need-to-know basis.

RQ2 *How to model and schedule the configuration process?* Starting from the original work on multi-level staged configuration, we extended the semantics of feature models with configuration paths. Configuration paths account for the dynamic nature of the configuration process. However, the theoretical and practical limitations of multi-level staged configura-

tion drove us to revise it entirely and propose a more expressive formalism supporting non-linear configuration processes. That new combined formalism is called *feature-based configuration workflow*, and is based on YAWL, a state-of-the-art workflow language. That work is motivated and illustrated through a configuration scenario taken from the aerospace industry.

RQ3 *How to ensure the satisfiability, safety and completion of the configuration process?* A formal semantics provides the compulsory basis for tool development and automation. However, careless design choices can yield unsatisfiable models, and inappropriate configuration decisions can prevent the correct completion of the configuration process. Safeguards are thus necessary on top of the semantics. To that end, we defined a satisfiability property that guarantees that at least one valid product can be built from the FCW, and verification mechanisms that ensure that any decision left open during the configuration can be made later. We then capitalised on the YAWL workflow engine and advances in SAT solvers to design efficient verification algorithms. The performance of our algorithms and the impact of workflow constructs was finally evaluated on 52 different feature-based configuration workflows.

RQ4 *How to handle conflictual user decisions?* Conflicts can occur at two levels: when a user alters previous decisions; and when multiple users concurrently configure the same feature model. We have detailed an algorithm that detects conflicts and generates fixes for user decisions. This algorithm relies on the known theory of diagnosis used in AI. We have discussed how this algorithm can be extended to support concurrent user configuration and reported preliminary results.

10.3 Perspective

Our contribution gave a robust foundation to the emerging requirements of collaborative feature-based configuration. Coming back to Thoreau's quote, we now have to firmly tie the castle onto that new foundation. This bond could be achieved by the development and integration of tool support. Enhanced tool support is the gateway to a complete and systematic empirical validation. Indeed, our experience and insight gained in Chapter 3 indicate that domain-specific frontends have the highest chance to receive the approbation of practitioners. Our early experiments focused mostly on the evaluation of the overall performance and correctness of our algorithms. The interface proposed in our toolset is more of a proof of concept rather than a customer-ready frontend. As such, it is too generic for practitioners to champion it. However, it paves the way for professional-grade tool support.

We discuss below in greater details the perspectives of this thesis along four dimensions. We first discuss the limitations of the feature modelling language itself and its impact on our definitions. Then, we successively explore avenues for further work on multi-view feature models, feature-based configuration workflows, and conflict management.

10.3.1 Expressiveness

Feature attributes are partially supported. In Section 2.3, we pointed out that *feature attributes* are not included in the semantics of feature models used in FCWs. They were re-introduced in Chapter 8 to provide comprehensive support for fix generation.

The addition of attributes to multi-view raises the question whether attributes should be considered independently of the feature to which they belong. If an attribute and its feature are an atomic unit, visualisation would not be affected. If they are distinct entities, then both the view specification and visualisations would have to be adapted. Empirical evaluations are needed to determine which alternative is best in practice and if the latter is chosen, how attributes should be presented if its feature is not in a view. In either case, Definition 5.3 will have to be adapted to take constraints over attributes into consideration. The addition of attributes would not alter the definition of feature-based configuration workflows as they consider views as the base unit. Verification would not be affected either, modulo the adaptation of Definition 5.3.

Feature cloning is not supported. *Feature cloning* (through *feature references* and $\langle i..j \rangle$ *feature cardinalities* with $j > 1$) was also excluded from the syntax and semantics of feature models. Feature cloning would require more drastic changes to the semantics, since the same feature might appear several times in a configuration. Besides a complete adaptation of the semantic domain, it would also require significant revision of the constraint system. As discussed in a preliminary extension of the semantics [MCHB11], future work includes the elaboration of an extended constraint language as well as more advanced reasoning algorithms due to the possibly infinite size of the domains to explore when performing formal analyses.

The integration with CVL is pending. Ultimately, we will also have to compose with CVL (Common Variability Standard [1]), the OMG attempt to standardise a variability modelling language. At the time of writing, the specification is still a draft and no official release has been issued. Yet, it is obvious that the implication of the leading tool vendors and

[1] `http://www.omgwiki.org/variability/`

some major industrial players will make it a reference in variability management. It is however impossible to tell now how much work will be needed to adapt our definitions to that standard.

The feature hierarchy can be optional. Our experience with Linux questioned the necessity of a feature hierarchy. Although the configuration modelling language (Kconfig) allows to nest options, it is not mandatory to explicitly specify the option hierarchy. In fact, the hierarchy is interpreted from the constraints in the Kconfig files to build the tree displayed in the configurator. This approach has been successfully used in hundreds of projects involving thousands of users. This raises the question whether the hierarchy should be syntactically captured in the model or left implicit. In the same vein, the orthogonal variability model only defines a hierarchy between a variation point and its variant. The relationship between variation points is established through crosscutting constraints.

10.3.2 Multi-view Feature Model

Alternative view specification techniques could be explored. Both existing techniques and our toolset use textual notations to define views. To our knowledge, no graphical or interactive technique has been proposed to build views on a feature model. This is an interesting topic for future investigation. Then, the pros and cons of the various techniques could be studied in greater depth (e.g. empirically), and their combinations could be envisaged.

Alternative view-rendering techniques could be explored. To provide more flexibility to the configuration environment and more precise contextual information to the user, we developed three visualisations. This improvement is, however, limited to tree-like representations of feature models. Recent advances deviate from the traditional explorer-like representations [BTN+08, CHBT10] while others recommend dedicated configuration interfaces [PBD10]. Understanding the most suitable interfaces for multi-view feature-based configuration in these techniques will require qualitative user studies.

10.3.3 Feature-based Configuration Workflow

Pre-defined configuration processes should be defined. Software configuration management taught us that genericity in configuration processes often increases development and maintenance costs. Final users usually prefer to reuse existing models at the expense of less flexibility. These models, however, rarely meet the needs of an organisation out of the box. Configurable workflows were thus proposed (e.g. [DRvdA+06,

GvdAJVLR07]) to allow a better alignment between off-the-shelf models and business requirements. They could be used during domain engineering, and then configured during application engineering. Assuming the workflow configuration stage occurs before configuration, no adaptation to our work would be required.

Declarative workflow specification could be explored. Alternatively to YAWL, declarative workflow modelling techniques like [DVALV10] could also be used. Declarative workflows specify a minimal set of constraints that must be satisfied during the execution. Unlike imperative workflow languages like YAWL or BPMN, the order in which the tasks are executed is not explicitly specified. It would be interesting to evaluate whether declarative workflows are better suited than imperative workflows to specify ordering constraints on configuration tasks. The relative simplicity of the configuration processes we observed might be an indicator in favour of declarative workflows.

The sensitivity of the safety analysis to *or*-joins should be reduced. The experiment with feature-based configuration workflows showed that the safety analysis was sensitive to *or*-joins. Furthermore, this limitation was independent of the size of the workflows. A quick fix for that limitation is to preclude the use of *or*-joins. Additional work is still needed to provide a more robust algorithm.

10.3.4 Conflict detection and resolution

Further experiments are ongoing. We are now working on the application of range fixes to Kconfig configurators. In addition, our industry partner has shown interest in including range fixes in their tool, and we are discussing the evaluation of our approach on large-scale industrial models and configurations.

Collaborative conflict resolution is work in progress. We discussed two open challenges for the application of range fix generation in collaborative environments. The first challenge is the resolution of direct conflicts, i.e., conflictual decisions about the same option. The second challenge is the integration of fix generation in existing frameworks such as those used in configuration management or commercial collaborative development platforms. Both challenges still require some work upstream fix generation to understand how teams deal with configuration conflicts.

Appendix

A

Greyed and Pruned Visualisations: Examples of Transformations

A.1 Pruned and Collapsed Visualisation of the eVoting example

Table A.1 synthesizes the results of the three transformations presented in Figure 5.3. The column of the greyed visualisation simply contains the features, decomposition edges and cardinalities of the FD. The boldfaced features are non-primitive features added to ensure the correctness of transformations (see Section 2.3).

In the pruned case, we see that the decomposition edges containing B, DO and DB have been pruned and removed from the list, and so are their associated cardinalities. The cardinalities that have have been recalculated are underlined. The new value $\lambda^p_{v_1}(A)$ is obtained with $\langle max(0,1-1)..min(3,3-1)\rangle$ whereas $\lambda^p_{v_1}(D)$ is $\langle max(0,1-2)..min(1,3-2)\rangle$. Note here that the minimum cardinality of D could have been negative, hence the need to set it to 0.

The only node removed in the pruned visualisation is A[1]. Which results in two collapsed nodes (i.e. Y and O). These nodes are directly connected to the root as their parent is pruned away. The cardinality of V must thus be recalculated, as detailed below.

$$
\begin{array}{l}
ms\_min^p_{v_1}(V) = \{mincard^c_{v_1}(\dot{A})\} \uplus \{1,1\} \\
mincard^c_{v_1}(V) = \sum min_3\{0\} \uplus \{1,1\} = 0+1+1 = 2 \\
ms\_max^p_{v_1}(V) = \{maxcard^c_{v_1}(\dot{A})\} \uplus \{1,1\} \\
maxcard^c_{v_1}(V) = \sum max_3\{2\} \uplus \{1,1\} = 1+1+2 = 4
\end{array}
$$

[1] And so is its parent non-primitive feature $\dot{A}$.

Table A.1 – Results of the calculation of the transformations on Figure 5.3.4.

Greyed			**Pruned**		
$N^g_{v_1}$	$DE^g_{v_1}$	$\lambda^g_{v_1}$	$N^p_{v_1}$	$DE^p_{v_1}$	$\lambda^p_{v_1}$
$\{ V,$ $\dot{\mathbf{E}},$ $E,$ $\dot{\mathbf{A}},$ $A,$ $Y,$ $O,$ $B,$ $\dot{\mathbf{D}},$ $D,$ $DY,$ $DO,$ DB $\}$	$\{ (\mathbf{V}, \dot{\mathbf{E}}),$ $(\dot{\mathbf{E}}, \mathbf{E}),$ $(\mathbf{V}, \dot{\mathbf{A}}),$ $(\dot{\mathbf{A}}, \mathbf{A}),$ $(A, Y),$ $(A, O),$ $(A, B),$ $(\mathbf{V}, \dot{\mathbf{D}}),$ $(\dot{\mathbf{D}}, \mathbf{D}),$ $(D, DY),$ $(D, DO),$ (D, DB) $\}$	$\lambda^g_{v_1}(V) = \langle 3..3 \rangle,$ $\lambda^{\mathbf{g}}_{\mathbf{v_1}}(\dot{\mathbf{E}}) = \langle \mathbf{0..1} \rangle,$ $\lambda^g_{v_1}(E) = \langle 0..0 \rangle,$ $\lambda^{\mathbf{g}}_{\mathbf{v_1}}(\dot{\mathbf{A}}) = \langle \mathbf{0..1} \rangle,$ $\lambda^g_{v_1}(A) = \langle 1..3 \rangle,$ $\lambda^g_{v_1}(Y) = \langle 0..0 \rangle,$ $\lambda^g_{v_1}(O) = \langle 0..0 \rangle,$ $\lambda^g_{v_1}(B) = \langle 0..0 \rangle,$ $\lambda^{\mathbf{g}}_{\mathbf{v_1}}(\dot{\mathbf{D}}) = \langle \mathbf{0..1} \rangle,$ $\lambda^g_{v_1}(D) = \langle 1..1 \rangle,$ $\lambda^g_{v_1}(DY) = \langle 0..0 \rangle,$ $\lambda^g_{v_1}(DO) = \langle 0..0 \rangle,$ $\lambda^g_{v_1}(DB) = \langle 0..0 \rangle$	$\{ V,$ $\dot{\mathbf{E}},$ $E,$ $\dot{\mathbf{A}},$ $A,$ $Y,$ $O,$ $\dot{\mathbf{D}},$ $D,$ DY $\}$	$\{ (\mathbf{V}, \dot{\mathbf{E}}),$ $(\dot{\mathbf{E}}, \mathbf{E}),$ $(\mathbf{V}, \dot{\mathbf{A}}),$ $(\dot{\mathbf{A}}, \mathbf{A}),$ $(A, Y),$ $(A, O),$ $(\mathbf{V}, \dot{\mathbf{D}}),$ $(\dot{\mathbf{D}}, \mathbf{D}),$ (D, DY) $\}$	$\lambda^p_{v_1}(V) = \langle 3..3 \rangle,$ $\lambda^{\mathbf{p}}_{\mathbf{v_1}}(\dot{\mathbf{E}}) = \langle \mathbf{0..1} \rangle,$ $\lambda^p_{v_1}(E) = \langle 0..0 \rangle,$ $\lambda^{\mathbf{p}}_{\mathbf{v_1}}(\dot{\mathbf{A}}) = \langle \mathbf{0..1} \rangle,$ $\lambda^p_{v_1}(A) = \langle 0..2 \rangle,$ $\lambda^p_{v_1}(Y) = \langle 0..0 \rangle,$ $\lambda^p_{v_1}(O) = \langle 0..0 \rangle,$ $\lambda^{\mathbf{p}}_{\mathbf{v_1}}(\dot{\mathbf{D}}) = \langle \mathbf{0..1} \rangle,$ $\lambda^p_{v_1}(D) = \langle 0..1 \rangle,$ $\lambda^p_{v_1}(DY) = \langle 0..0 \rangle$

Collapsed		
$N^c_{v_1}$	$DE^c_{v_1}$	$\lambda^c_{v_1}$
$\{ V,$ $\dot{\mathbf{E}},$ $E,$ $Y,$ $O,$ $\dot{\mathbf{D}},$ $D,$ DY $\}$	$\{ (\mathbf{V}, \dot{\mathbf{E}}),$ $(\dot{\mathbf{E}}, \mathbf{E}),$ $(A, Y),$ $(A, O),$ $(\mathbf{V}, \dot{\mathbf{D}}),$ $(\dot{\mathbf{D}}, \mathbf{D}),$ (D, DY) $\}$	$\lambda^c_{v_1}(V) = \langle 2..4 \rangle,$ $\lambda^{\mathbf{c}}_{\mathbf{v_1}}(\dot{\mathbf{E}}) = \langle \mathbf{0..1} \rangle,$ $\lambda^c_{v_1}(E) = \langle 0..0 \rangle,$ $\lambda^c_{v_1}(Y) = \langle 0..0 \rangle,$ $\lambda^c_{v_1}(O) = \langle 0..0 \rangle,$ $\lambda^{\mathbf{c}}_{\mathbf{v_1}}(\dot{\mathbf{D}}) = \langle \mathbf{0..1} \rangle,$ $\lambda^c_{v_1}(D) = \langle 0..1 \rangle,$ $\lambda^c_{v_1}(DY) = \langle 0..0 \rangle$

$$
\begin{aligned}
ms\_min^p_{v_1}(\dot{A}) &= \{mincard^c_{v_1}(A)\} \uplus \{\} \\
mincard^c_{v_1}(\dot{A}) &= \textstyle\sum min_0\{0\} = 0 \\
ms\_max^p_{v_1}(\dot{A}) &= \{maxcard^c_{v_1}(A)\} \uplus \{\} \\
maxcard^c_{v_1}(\dot{A}) &= \textstyle\sum max_1\{2\} = 2
\end{aligned}
$$

$$
\begin{aligned}
ms\_min^p_{v_1}(A) &= \{mincard^c_{v_1}(B)\} \uplus \{1, 1\} \\
mincard^c_{v_1}(A) &= \textstyle\sum min_1\{0\} \uplus \{1, 1\} = 0 \\
ms\_max^p_{v_1}(A) &= \{maxcard^c_{v_1}(B)\} \uplus \{1, 1\} \\
maxcard^c_{v_1}(A) &= \textstyle\sum max_3\{0\} \uplus \{1, 1\} = 0 + 1 + 1 = 2
\end{aligned}
$$

$$
\begin{aligned}
ms\_min^p_{v_1}(B) &= \{\} \uplus \{\} \\
mincard^c_{v_1}(B) &= \textstyle\sum min_0\{\} = 0 \\
ms\_max^p_{v_1}(B) &= \{\} \uplus \{\} \\
maxcard^c_{v_1}(B) &= \textstyle\sum max_0\{\} = 0
\end{aligned}
$$

The cardinality of D is the same as in the pruned visualisation:

$$
\begin{aligned}
ms\_min^{p}_{v_1}(D) &= \{mincard^{c}_{v_1}(DO), mincard^{c}_{v_1}(DB)\} \uplus \{1\} \\
mincard^{c}_{v_1}(D) &= \sum min_1\{0,0\} \uplus \{1\} = 0 \\
ms\_max^{p}_{v_1}(D) &= \{maxcard^{c}_{v_1}(DO), maxcard^{c}_{v_1}(DB)\} \uplus \{1\} \\
maxcard^{c}_{v_1}(D) &= \sum max_1\{0,0\} \uplus \{1\} = 1
\end{aligned}
$$

$$
\begin{aligned}
ms\_min^{p}_{v_1}(DO) &= \{\} \uplus \{\} \\
mincard^{c}_{v_1}(DO) &= \sum min_0\{\} = 0 \\
ms\_max^{p}_{v_1}(DO) &= \{\} \uplus \{\} \\
maxcard^{c}_{v_1}(DO) &= \sum max_0\{\} = 0
\end{aligned}
$$

$$
\begin{aligned}
ms\_min^{p}_{v_1}(DB) &= \{\} \uplus \{\} \\
mincard^{c}_{v_1}(DB) &= \sum min_0\{\} = 0 \\
ms\_max^{p}_{v_1}(DB) &= \{\} \uplus \{\} \\
maxcard^{c}_{v_1}(DB) &= \sum max_0\{\} = 0
\end{aligned}
$$

A.1.1 Pruned and collapsed visualisation of the PloneMeeting Manager

The pruned and collapsed visualisations of the PloneMeeting Manager are presented in Figure A.1. Although calculated differently, the cardinalities of both visualisations are all equal.

Pruned The new cardinalities of the pruned visualisation are calculated as follows:

$$
\begin{aligned}
\lambda^{p}_{PM}(MC) &= \langle max(0, 7-1)..min(7, 7-1)\rangle = \langle 6..6\rangle \\
\lambda^{p}_{PM}(D) &= \langle max(0, 3-1)..min(3, 3-1)\rangle = \langle 2..2\rangle \\
\lambda^{p}_{PM}(W) &= \langle max(0, 3-2)..min(3, 3-2)\rangle = \langle 1..1\rangle \\
\lambda^{p}_{PM}(T) &= \langle max(0, 2-1)..min(2, 2-1)\rangle = \langle 1..1\rangle
\end{aligned}
$$

Collapsed The new cardinalities of the collapsed visualisation are calculated as follows:

$$
\begin{aligned}
mincard^{c}_{PM}(MC) &= \sum min_7\{0\} \uplus \{1,1,1,1,1,1\} = 6 \\
maxcard^{c}_{PM}(MC) &= \sum max_7\{0\} \uplus \{1,1,1,1,1,1\} = 6 \\
\lambda^{c}_{PM}(MC) &= \langle 6..6\rangle
\end{aligned}
$$

$$
\begin{aligned}
mincard^{c}_{PM}(D) &= \sum min_3\{0\} \uplus \{1,1\} = 2 \\
maxcard^{c}_{PM}(D) &= \sum max_3\{0\} \uplus \{1,1\} = 2 \\
\lambda^{c}_{PM}(D) &= \langle 2..2\rangle
\end{aligned}
$$

$$
\begin{aligned}
mincard^{c}_{PM}(W) &= \sum min_3\{0,0\} \uplus \{1\} = 1 \\
maxcard^{c}_{PM}(W) &= \sum max_3\{0,0\} \uplus \{1\} = 1 \\
\lambda^{c}_{PM}(W) &= \langle 1..1\rangle
\end{aligned}
$$

$$
\begin{aligned}
mincard^{c}_{PM}(T) &= \sum min_2\{0\} \uplus \{1\} = 1 \\
maxcard^{c}_{PM}(T) &= \sum max_2\{0\} \uplus \{1\} = 1 \\
\lambda^{c}_{PM}(T) &= \langle 1..1\rangle
\end{aligned}
$$

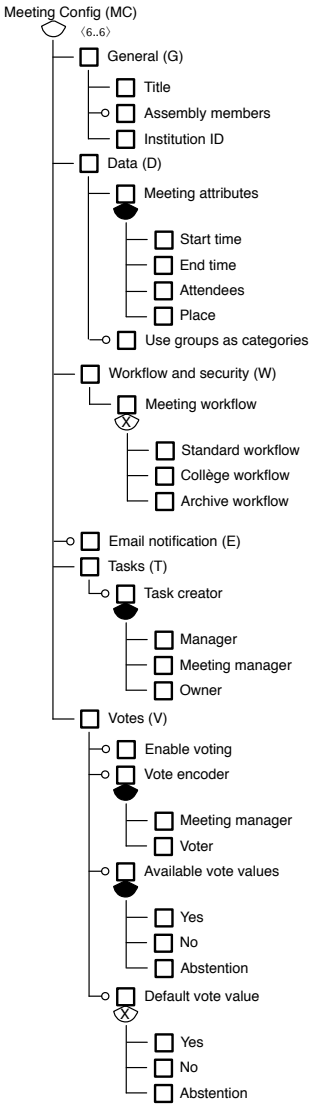
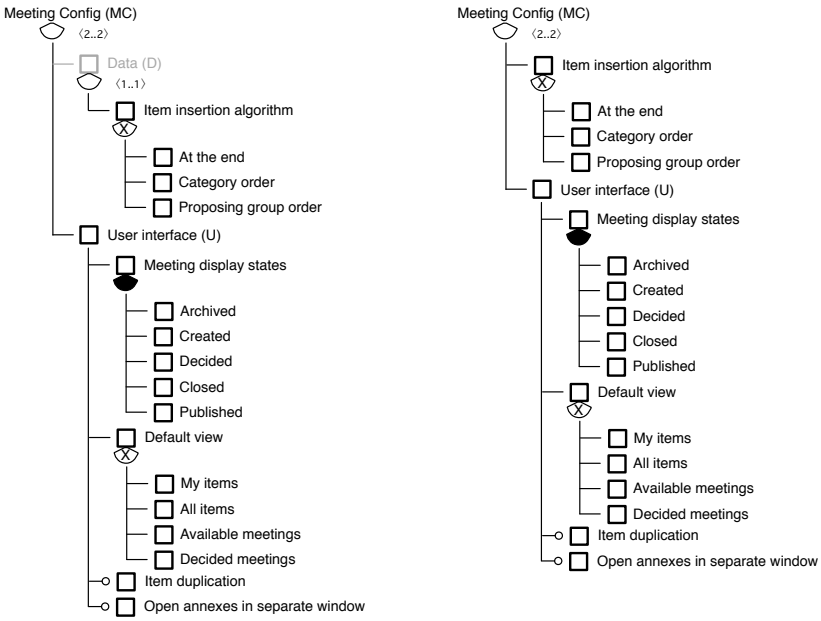


Figure A.1 – Pruned and collapsed visualisation of the PloneMeeting manager view.

A.1.2 Prune and collapsed visualisations of the User

Pruned The pruned visualisation of the User is presented in Figure A.2(a). The new cardinalities are calculated as follows:

$$\lambda^{p}_{User}(MC) = \langle max(0, 7-5)..min(7, 7-5)\rangle = \langle 2..2\rangle$$
$$\lambda^{p}_{User}(D) = \langle max(0, 3-2)..min(3, 3-2)\rangle = \langle 1..1\rangle$$



(a) Pruned visualisation (b) Collapsed visualisation

Figure A.2 – Pruned and collapsed visualisation of the user view.

Collapsed The collapsed visualisation of the PloneMeeting manager is presented in Figure A.2(b). The new cardinalities are calculated as follows:

$$
\begin{array}{rcl}
mincard^c_{User}(D) & = & \sum min_3\{0,0\} \uplus \{1\} = 1 \\
ms\_min^c_{User}(MC) & = & \{mincard^c_{User}(G), mincard^c_{User}(D), mincard^c_{User}(W), \\
 & & mincard^c_{User}(U), mincard^c_{User}(E), mincard^c_{User}(T), \\
 & & mincard^c_{User}(V)\} \uplus \{1\} \\
 & = & \{0,1,0,0,0,0\} \uplus \{1\} \\
mincard^c_{User}(MC) & = & \sum min_7\{0,1,0,0,0,0\} \uplus \{1\} = 2
\end{array}
$$

$$
\begin{array}{rcl}
maxcard^c_{User}(D) & = & \sum max_3\{0,0\} \uplus \{1\} = 1 \\
ms\_max^c_{User}(MC) & = & \{maxcard^c_{User}(G), maxcard^c_{User}(D), maxcard^c_{User}(W), \\
 & & maxcard^c_{User}(U), maxcard^c_{User}(E), maxcard^c_{User}(T), \\
 & & maxcard^c_{User}(V)\} \uplus \{1\} \\
 & = & \{0,1,0,0,0,0\} \uplus \{1\} \\
maxcard^c_{User}(MC) & = & \sum max_7\{0,1,0,0,0,0\} \uplus \{1\} = 2 \\
\lambda^c_{User}(MC) & = & \langle 2..2 \rangle
\end{array}
$$

Appendix

B

Detailed Example of $[\![.]\!]_{\text{MLSC}}$ and Proof Helper

B.1 Calculation Details of Figure 6.6

This section presents the calculation details of the two configuration paths, π_i and π_j, used in the example of Section 6.3. In the remainder of this section, we will use d when referring to the FM of Figure 6.5 and we note $\sigma \overset{L_j}{\rightarrow} \sigma'$ if $\pi = ..\sigma\sigma'..$ and $\sigma' \in \Sigma_j$.

B.1.1 Legal path (π_i)

This configuration path, shown in Figure 6.3 and in the middle part of Figure 6.6, actually corresponds to the one used in the original illustration by Czarnecki *et al.* [CHE05], which was summarised in Section 6.1. One can see in Figure 6.3 that it consists of two manual configuration stages and one automatic specialisation stage. Since the *inter-level links* already implement the latter stage, there is no need to represent it as a dedicated transition between two stages. The results of the two former stages result from manual configurations that cannot be automatically derived from the constraint set and thus have to be represented by two separate configuration sets, *viz.* σ_{2_i} and σ_{3_i}.

In terms of $[\![d]\!]_{CP}$, π_i is a sequence $\sigma_1\sigma_{2_i}\sigma_{3_i}$ with:

$$
\begin{array}{lcll}
\sigma_1 & = & \{ & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, T_2, M_2, O_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, T_2, M_2, I_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, E_2, D_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2, B_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, G_2, T_2, M_2, O_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, G_2, T_2, M_2, I_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, G_2, E_2, D_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}, G_2, T_2, M_2, O_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}, G_2, T_2, M_2, I_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}, G_2, S_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}, G_2, S_2, B_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, E_2, D_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2, B_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{T_1}, G_2, T_2, M_2, O_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{T_1}, G_2, T_2, M_2, I_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{E_1}, G_2, E_2, D_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{S_1}, G_2, S_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{S_1}, G_2, S_2, B_2\} \quad \} \\
\sigma_{2_i} & = & \{ & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, E_2, D_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2\}, \\
 & & & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2, B_2\} \quad \} \\
\sigma_{3_i} & = & \{ & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, E_2, D_2\} \quad \}
\end{array}
$$

Note here that $|(\sigma_{3_i})| = 1$, which denotes the end of the configuration path (Definition 6.3), resulting in the single product $\{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, E_2, D_2\}$.

In order to show that $\pi_i \in [\![d]\!]_{\text{MLSC}}$, we have to find a level arrangement that satisfies rules (6.5.1) and (6.5.2). Let us examine each possible level arrangement in turn.

(a) For $\Sigma_{1_i} = \epsilon$ and $\Sigma_{2_i} = \sigma_{2_i}\sigma_{3_i}$, we have

$$|final(\Sigma_{1_i})_{|L_1}| = |\sigma_{1\,|L_1}| = 7 > \mathbf{1}$$

The arrangement is thus **rejected** because (6.5.1) evaluates to false.

(b) For $\Sigma_1 = \sigma_{2_i}\sigma_{3_i}$ and $\Sigma_{2_i} = \epsilon$, we have

$$|final(\Sigma_{1_i})_{|L_1}| = |\sigma_{3_i\,|L_1}| = 1$$

$$|final(\Sigma_{2_i})_{|L_2}| = |\sigma_{3_i\,|L_2}| = 1$$

and (6.5.1) evaluates to true. We have for $\sigma_1 \xrightarrow{L_1} \sigma_{2_i}$:

$$\begin{array}{lll}
\textbf{1.}\ \sigma_{\mathbf{1}} \setminus \sigma_{\mathbf{2_i}} = & \{ & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, T_2, M_2, O_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, T_2, M_2, I_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, E_2, D_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2, B_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, G_2, T_2, M_2, O_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, G_2, T_2, M_2, I_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, G_2, E_2, D_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}, G_2, T_2, M_2, O_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}, G_2, T_2, M_2, I_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}, G_2, S_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}, G_2, S_2, B_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, G_2, T_2, M_2, O_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, G_2, T_2, M_2, I_2\}, \\
& & \{\mathbf{G_1}, \mathbf{E_1}, G_2, E_2, D_2\}, \\
& & \{\mathbf{G_1}, \mathbf{S_1}, G_2, S_2\}, \\
& & \{\mathbf{G_1}, \mathbf{S_1}, G_2, S_2, B_2\} \quad \} \\
\textbf{2.}\ (\sigma_{\mathbf{1}} \setminus \sigma_{\mathbf{2_i}})_{|\mathbf{L_1}} = & \{ & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{E_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{S_1}\} \quad \}
\end{array}$$

and

$$\begin{array}{lll}
\textbf{1.}\ \sigma_{\mathbf{1}\,|\mathbf{L_1}} = & \{ & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{E_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{S_1}\} \quad \} \\
\textbf{2.}\ \sigma_{\mathbf{2_i}\,|\mathbf{L_1}} = & \{ & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}\} \quad \} \\
\textbf{3.}\ \sigma_{\mathbf{1}\,|\mathbf{L_1}} \setminus \sigma_{\mathbf{2_i}\,|\mathbf{L_1}} = & \{ & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{E_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{S_1}\} \quad \}
\end{array}$$

such that $(\sigma_1 \setminus \sigma_{2_i})_{|L_1} \subseteq (\sigma_{1\,|L_1} \setminus \sigma_{2_i\,|L_1})$ is true and we have for $\sigma_{2_i} \xrightarrow{L_1} \sigma_{3_i}$:

$$\begin{array}{lll}
(\sigma_{\mathbf{2_i}} \setminus \sigma_{\mathbf{3_i}})_{|\mathbf{L_1}} = & \{ & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2\}, \\
& & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2, B_2\} \quad \} \\
\sigma_{\mathbf{2_i}\,|\mathbf{L_1}} \setminus \sigma_{\mathbf{3_i}\,|\mathbf{L_1}} = & \emptyset &
\end{array}$$

The arrangement is thus **rejected** because (6.5.2) evaluates to false since $(\sigma_{2_i} \setminus \sigma_{3_i})_{|L_1} \subseteq (\sigma_{2_i\,|L_1} \setminus \sigma_{3_i\,|L_1})$ does not hold. This rejection is obvious since level L_1 was already fully configured at σ_{2_i}, hence the absence of deletable configurations from σ_{2_i} in L_1.

(c) For $\Sigma_{1_i} = \sigma_{2_i}$ and $\Sigma_{2_i} = \sigma_{3_i}$, we have

$$|final(\Sigma_{1_i})_{|L_1}| = |\sigma_{2_i\,|L_1}| = 1$$

$$|final(\Sigma_{2_i})_{|L_2}| = |\sigma_{3_i|L_2}| = 1$$

and (6.5.1) evaluates to true. We know from above that for $\sigma_1 \xrightarrow{L_1} \sigma_{2_i}$:

$$(\sigma_1 \setminus \sigma_{2_i})_{|L_1} \subseteq (\sigma_{1|L_1} \setminus \sigma_{2_i|L_1})$$

and we have for $\sigma_{2_i} \xrightarrow{L_2} \sigma_{3_i}$:

$$\begin{array}{lll} (\sigma_{\mathbf{2_i}} \setminus \sigma_{\mathbf{3_i}})_{|\mathbf{L_2}} = & \{ & \{G_2, S_2\}, \\ & & \{G_2, S_2, B_2\} \quad \} \\ \sigma_{\mathbf{2_i}|\mathbf{L_2}} \setminus \sigma_{\mathbf{3_i}|\mathbf{L_2}} = & \{ & \{G_2, S_2\}, \\ & & \{G_2, S_2, B_2\} \quad \} \end{array}$$

i.e. that $(\sigma_{2_i} \setminus \sigma_{3_i})_{|L_2} \subseteq (\sigma_{2_i|L_2} \setminus \sigma_{3_i|L_2})$. Since both (6.5.1) and (6.5.2) evaluate to true, the arrangement is **accepted**.

One level arrangement, *viz.* $\Sigma_{1_i} = \sigma_{2_i}$ and $\Sigma_{2_i} = \sigma_{3_i}$, satisfies rules (6.5.1) and (6.5.2) and thus $\pi_i \in [\![d]\!]_{\text{MLSC}}$.

B.1.2 Illegal path (π_j)

Let us now illustrate the illegal configuration path $\pi_j \in [\![d]\!]_{CP}$ sketched in Figure 6.6. Intuitively, π_j is illegal because feature B_2, belonging to L_2, is deselected even though L_1 is not yet finished. The configuration path π_j is thus the same as π_i except for σ_{2_j}, which is now:

$$\begin{array}{lll} \sigma_{2_j} = & \{ & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, E_2, D_2\}, \\ & & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2\} \quad \} \end{array}$$

Again, in order for $\pi_j \in [\![d]\!]_{\text{MLSC}}$ there must be at least one level arrangement that satisfies rules (6.5.1) and (6.5.2). Let us examine each possible arrangement in turn.

(a) For $\Sigma_{1_j} = \epsilon$ and $\Sigma_{2_j} = \sigma_{2_j}\sigma_{3_j}$, we have

$$|final(\Sigma_{1_j})_{|L_1}| = |\sigma_{1|L_1}| = 7 \quad > \mathbf{1}$$

The arrangement is thus **rejected** because (6.5.1) evaluates to false.

(b) For $\Sigma_{1_j} = \sigma_{2_j}\sigma_{3_j}$ and $\Sigma_{2_j} = \epsilon$, we have

$$|final(\Sigma_{1_j})_{|L_1}| = |\sigma_{3_j|L_1}| = 1$$

$$|final(\Sigma_{2_j})_{|L_2}| = |\sigma_{3_j|L_2}| = 1$$

and (6.5.1) evaluates to true. We have for $\sigma_1 \xrightarrow{L_1} \sigma_{2_j}$:

$$
\begin{aligned}
(\sigma_{\mathbf{1}} \setminus \sigma_{\mathbf{2_j}})_{|\mathbf{L_1}} = \quad \{ \quad & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}\}, \\
& \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}\}, \\
& \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}\}, \\
& \underline{\{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}\}}, \\
& \{\mathbf{G_1}, \mathbf{T_1}\}, \\
& \{\mathbf{G_1}, \mathbf{E_1}\}, \\
& \{\mathbf{G_1}, \mathbf{S_1}\} \quad \} \\
(\sigma_{\mathbf{1}} \setminus \sigma_{\mathbf{2_j}})_{|\mathbf{L_1}} = \quad & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}\}, \\
& \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}\}, \\
& \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}\}, \\
& \{\mathbf{G_1}, \mathbf{T_1}\}, \\
& \{\mathbf{G_1}, \mathbf{E_1}\}, \\
& \{\mathbf{G_1}, \mathbf{S_1}\} \quad \}
\end{aligned}
$$

The arrangement is thus **rejected** because (6.5.2) evaluates to false since $(\sigma_1 \setminus \sigma_{2_j})_{|L_1} \supset (\sigma_{1\,|L_1} \setminus \sigma_{2_j\,|L_1})$. Intuitively, this condition is false because the configuration $\underline{\{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2, B_2\}}$ is part of the deleted ones but was not deletable since B_2 belongs to another level and all configurations containing $\{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}\}$ where bound to be selected.

(c) For $\Sigma_{1_j} = \sigma_{2_j}$ and $\Sigma_{2_j} = \sigma_{3_j}$, we have

$$|final(\Sigma_{1_j})_{|L_1}| = |\sigma_{2_j\,|L_1}| = 1$$

$$|final(\Sigma_{2_j})_{|L_2}| = |\sigma_{3_j\,|L_2}| = 1$$

and (6.5.1) evaluates to true. Like for $\sigma_{2_j}, \sigma_{3_j} \in \Sigma_{1_j}$ and $\Sigma_{2_j} = \emptyset$, we have for $\sigma_1 \xrightarrow{L_1} \sigma_{2_j}$ that $(\sigma_1 \setminus \sigma_{2_j})_{|L_1} \supset (\sigma_{1\,|L_1} \setminus \sigma_{2_j\,|L_1})$. The arrangement is thus **rejected** because (6.5.2) evaluates to false.

Since none of the possible arrangements satisfies both (6.5.1) and (6.5.2), the configuration path π_j does not belong to $[\![d]\!]_{\text{MLSC}}$.

B.2 Proof Helper for Theorem B.1

The following theorem is used for the proof of Theorem B.1, Section 6.4. Basically, it says that when two sets of sets are reduced to include only sets containing certain elements and then subtracted, the result is included in (i.e. smaller than) the set obtained by subtracting first and reducing afterwards. Intuitively, the result of a subtraction operation is smaller the more elements in both sets "match up", and if a reduction is applied prior to subtracting, it becomes "more likely" for elements to match up, meaning that the result of the subtraction can be smaller than if the reduction was applied afterwards.

Theorem B.1 **Inclusion of deletable decisions in deleted decision set**

For some set N, if $\sigma, \sigma' \in \mathcal{PP}N$ so that $\sigma \supset \sigma'$ and $L \subseteq N$, then

$$\big\{\{c \cap L | c \in \sigma\} \setminus \{c \cap L | c \in \sigma'\}\big\} \subseteq \{c \cap L | c \in (\sigma \setminus \sigma')\}.$$

Proof.

$$\begin{aligned}
&\big\{\{c \cap L | c \in \sigma\} \setminus \{c \cap L | c \in \sigma'\}\big\} \\
=\ &\big\{a | (a \in \{c \cap L | c \in \sigma\}) \wedge (a \notin \{c \cap L | c \in \sigma'\})\big\} \\
=\ &\big\{a | (a \in \{p | p = \{x | \exists c \in \mathcal{P}N \bullet x \in c \wedge x \in L \wedge c \in \sigma\}\} \\
&\quad \wedge\ a \notin \{p | p = \{x | \exists c \in \mathcal{P}N \bullet x \in c \wedge x \in L \wedge c \in \sigma'\}\})\big\} \\
=\ &\big\{p | p = \{x | (\exists c \in \mathcal{P}N \bullet x \in c \wedge x \in L \wedge c \in \sigma) \\
&\qquad \wedge \neg(\exists c \in \mathcal{P}N \bullet x \in c \wedge x \in L \wedge c \in \sigma')\}\big\} \\
=\ &\big\{p | p = \{x | (\exists c \in \mathcal{P}N \bullet x \in c \wedge x \in L \wedge c \in \sigma) \\
&\qquad \wedge\ \ (\forall c \in \mathcal{P}N \bullet \neg(x \in c \wedge x \in L \wedge c \in \sigma'))\}\big\} \\
\subseteq\ &\\
&\big\{p | p = \{x | \exists c \in \mathcal{P}N \bullet (x \in c \wedge x \in L \wedge c \in \sigma) \\
&\qquad\qquad\qquad \wedge \neg(x \in c \wedge x \in L \wedge c \in \sigma')\}\big\} \\
=\ &\big\{p | p = \{x | \exists c \in \mathcal{P}N \bullet x \in c \wedge x \in L \wedge c \in \sigma \wedge \neg(c \in \sigma')\}\big\} \\
=\ &\big\{p | p = \{x | x \in c \wedge x \in L \wedge c \in \sigma \wedge c \notin \sigma'\}\big\} \\
=\ &\big\{c \cap L | c \in \sigma \wedge c \notin \sigma'\big\} \\
=\ &\big\{c \cap L | c \in (\sigma \setminus \sigma')\big\}
\end{aligned}$$

□

Appendix

C

Safety Analysis: Workflow Transformation Algorithms

Algorithm 2 Transformation of the workflow of a given FCW m.

Require: w is (weak) sound

1: **function** TRANSFORMFCW(m)
2: $transformed \leftarrow \emptyset$
3: **for all** $v \in V$ **do**
4: $tw \leftarrow w$
5: UPDATEBEGINCONDITION(tw)
6: BUILDENDCHECK(tw)
7: UPDATEENDCONDITION(tw, v)
8: BUILDCONDITIONCHECK($tw, stop(v)$)
9: BUILDPRESENCECHECK($tw, stop(v)$)
10: BUILDTASKCHECK($tw, start(v), stop(v)$)
11: BUILDCLEAN($tw, stop(v)$)
12: $transformed \leftarrow tw \cup transformed$
13: **end for**
14: **return** *transformed*
15: **end function**

Algorithm 3 Update the begin condition

1: **function** UPDATEBEGINCONDITION(w)
2: $T \leftarrow T \cup init$
3: $split(init) \leftarrow AND$
4: $first \leftarrow f | (i, f) \in F$
5: $F \leftarrow F \setminus (i, first)$
6: $F \leftarrow F \cup (i, init) \cup (init, first)$
7: **end function**

Algorithm 4 Build the end check

1: **function** BUILDENDCHECK(w)
2: $T \leftarrow T \cup check\_out \cup exit$
3: $C \leftarrow C \cup end$
4: $split(check\_out) \leftarrow AND$
5: $join(exit) \leftarrow AND$
6: $lasts \leftarrow l | (l, e) \in F$
7: **for all** $l \in lasts$ **do**
8: $F \leftarrow F \setminus (l, e)$
9: $F \leftarrow F \cup (l, end)$
10: **end for**
11: $F \leftarrow F \cup (end, check\_out) \cup (exit, e)$
12: **end function**

Algorithm 5 Update the end condition

1: **function** UPDATEENDCONDITION(w,v)
2: **if** $stop(v) = e$ **then**
3: $stop(v) \leftarrow end$
4: **end if**
5: **end function**

Algorithm 6 Build the condition check

function BUILDCONDITIONCHECK(w, c)
  $predecessors \leftarrow p | (p, c) \in F$
  $C \leftarrow C \cup subs\_c$
  $T \leftarrow T \cup sync\_c \cup run\_c$
  $split(sync\_c) = AND$
  $split(run\_c) = AND$
  $join(run\_c) = AND$
  $F \leftarrow F \cup (subs\_c, sync\_c) \cup (run\_c, c)$
  for all $p \in predecessors$ **do**
    $F \leftarrow F \setminus (p, c)$
    $F \leftarrow F \cup (p, subs\_c)$
  end for
end function

Algorithm 7 Build the presence check

function BUILDPRESENCECHECK(w, c)
  $C \leftarrow C \cup check\_out\_c \cup dead\_c \cup active\_task\_c \cup active\_c$
  $T \leftarrow T \cup check\_active\_c \cup proceed\_c \cup end\_c \cup exit\_c$
  $join(check\_active\_c) = AND$
  $join(proceed\_c) = AND$
  $join(end\_c) = AND$
  $join(exit\_c) = XOR$
  $F \leftarrow F \cup (check\_out, check\_out\_c) \cup (check\_out\_c, check\_active\_c) \cup (check\_out\_c, proceed\_c)$
  $F \leftarrow F \cup (proceed\_c, exit\_c) \cup (check\_active\_c, end\_c) \cup (end\_c, exit\_c) \cup (active\_task\_c, check\_active\_c) \cup (active\_c, end\_c)$
  $F \leftarrow F \cup (dead\_c, proceed\_c) \cup (init, dead\_c) \cup (run\_c, active\_c) \cup (exit\_c, exit)$
end function

Algorithm 8 Build the task check

function BUILDTASKCHECK(w, t, c)
  $predecessors \leftarrow p | (p,t) \in F$
  $successors \leftarrow s | (t,s) \in F$
  $tsplit \leftarrow split(t)$
  $tjoin \leftarrow join(t)$
  $C \leftarrow C \cup enabled\_t \cup active\_t \cup disabled\_t \cup check\_c\_t$
  $T \leftarrow T \cup run\_t \cup proceed\_t \cup validate\_t \cup disable\_t \cup end\_c\_t \cup$
  $join(run\_t) = tjoin$
  $split(t) \leftarrow AND$
  $join(t) \leftarrow AND$
  $split(proceed\_t) = tsplit$
  $split(validate\_t) \leftarrow AND$
  $join(validate\_t) \leftarrow AND$
  $join(disable\_t) \leftarrow AND$
  $split(disable\_t) \leftarrow AND$
  $join(end\_c\_t) \leftarrow XOR$
  for all $p \in predecessors$ **do**
    $F \leftarrow F \setminus (p,t)$
    $F \leftarrow F \cup (p, run\_t)$
  end for
  for all $s \in successors$ **do**
    $F \leftarrow F \setminus (t,s)$
    $F \leftarrow F \cup (proceed\_t, s)$
  end for
  $F \leftarrow F \cup (init, enabled\_t) \cup (init, disabled\_t)$
  $F \leftarrow F \cup (run\_t, t) \cup (enabled\_t, t)$
  $F \leftarrow F \cup (t, enabled\_t) \cup (t, proceed\_t) \cup (t, active\_t) \cup (t, active\_task\_c)$
  $F \leftarrow F \cup (disabled\_t, disable\_t) \cup (active\_t, validate\_t) \cup (sync\_c, check\_c\_t)$
  $F \leftarrow F \cup (check\_c\_t, validate\_t) \cup (check\_c\_t, disable\_t) \cup (end\_c\_t, run\_c)$
  $F \leftarrow F \cup (validate\_t, end\_c\_t) \cup (validate\_t, active\_t)$
  $F \leftarrow F \cup (disable\_t, end\_c\_t) \cup (disable\_t, disabled\_t)$
  $rem(t) \leftarrow rem(t) \cup dead\_c \cup disabled\_c$
  $rem(disable\_t) \leftarrow rem(disable\_t) \cup enabled\_c \cup disabled\_t$
  $rem(validate\_t) \leftarrow rem(validate\_t) \cup active\_t$
end function

Algorithm 9 Build the clean check

1: **function** BUILDCLEAN(w,c)
2: $rem(exit) \leftarrow rem(exit) \cup active\_task\_c \cup check\_out\_c \cup active\_c \cup check\_active\_c \cup dead\_c \cup proceed\_c \cup end\_c \cup exit\_c$
3: **end function**

Index

Symbols

A

B

C

D

E

F

K

L

M

W

Y

Bibliography

[AC04] M. Antkiewicz and K. Czarnecki. Featureplugin: feature modeling plug-in for eclipse. In *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72, New York, NY, USA, 2004. ACM.

[AC06] F. Ahmed and L. F. Capretz. Maturity assessment framework for business dimension of software product family. *IBIS*, 1(1):9–32, 2006.

[ACLF09] M. Acher, P. Collet, P. Lahire, and R. France. Composing Feature Models. In *Proceedings of the 2nd International Conference on Software Language Engineering (SLE'09)*, LNCS, pages 62–81. Springer, 2009.

[ACLF10] M. Acher, P. Collet, P. Lahire, and R. France. Managing variability in workflow with feature model composition operators. In B. Baudry and E. Wohlstadter, editors, *Software Composition*, volume 6144 of *Lecture Notes in Computer Science*, pages 17–33. Springer Berlin / Heidelberg, 2010.

[AD11] J.-M. Astesana and A. Dauron. Spécification et configuration de la ligne de produits véhicule de renault. In *Journée Lignes de Produits*, Paris, October 2011.

[Ada10] Michael Adams. Yawl - user manual, version 2.1. *The YAWL Foundation*, 2010.

[AdOM+09] M. Anastasopoulos, T. H. B. de Oliveira, D. Muthig, E. S. Almeida, and S. R. de Lemos Meira. Structruring the product line modeling space: Strategies and examples. In *Proceedings of the 3rd International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'09)*, Sevilla, Spain, January 2009. University of Duisburg-Essen.

[AJL+10] A. Abele, R. Johansson, H. Lo, Y. Papadopoulos, M.-O. Reiser, D. Servat, M. Torngren, and M. Weber. The cvm framework - a prototype tool for compositional variability management. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10)*, pages 101–105, Linz, Austria, 2010. University of Duisburg-Essen.

[Aoy05] M. Aoyama. Persona-and-scenario based requirements engineering for software embedded in digital consumer products. In *Proceedings of the 13th International Conference on Requirements Engineering (RE'05)*, pages 85–94, Paris, France, 2005. IEEE Computer Society.

[Apa11] Apache. Apache subversion. http://subversion.apache.org/, October 2011.

[ASM04] T. Asikainen, T. Soininen, and T. Mannisto. A koala-based approach for modelling and deploying configurable software product families. In Frank van der Linden, editor, *Software Product-Family Engineering*, volume 3014 of *Lecture Notes in Computer Science*, pages 225–249. Springer Berlin / Heidelberg, 2004.

[Bab86] W. A. Babich. *Software configuration management: coordination for team productivity.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Bat05] D. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Software Product Line Conference (SPLC'05)*, pages 7–20, Rennes, France, 2005. Springer Berlin / Heidelberg.

[BBRC06] D. Batory, D. Benavides, and A. Ruiz-Cortes. Automated analysis of feature models: challenges ahead. *Communication of the ACM*, 49(12):45–47, 2006.

[BC89] K. Beck and W. Cunningham. A laboratory for teaching object oriented thinking. In *Proceedings of the Conference on*

Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89), pages 1–6, New York, NY, USA, 1989. ACM.

[BCE+06] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A manifesto for model merging. In *Proceedings of the International workshop on Global integrated model management (GaMMa '06)*, pages 5–12, New York, NY, USA, 2006. ACM.

[BCFH10] Q. Boucher, A. Classen, P. Faber, and P. Heymans. Introducing TVL, a text-based feature modelling language. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10)*, pages 159–162, Linz, Austria, 2010. University of Duisburg-Essen.

[BCH+10] Q. Boucher, A. Classen, P. Heymans, A. Bourdoux, and L. Demonceau. Tag and prune: a pragmatic approach to software product line implementation. In *Proceedings of the 25th international conference on Automated software engineering (ASE'10)*, pages 333–336, New York, NY, USA, 2010. ACM.

[BCK03] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, Second Edition.* Addison-Wesley Professional, 2003. Chapter 3.

[BCM+03] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003.

[BCW10] K. Bąk, K. Czarnecki, and Andrzej W. Feature and meta-models in clafer: Mixed, specialized, and coupled. In *Proceedings of the 3rd International Conference on Software Language Engineering (SLE'10)*, pages 102–122, Eindhoven, The Netherlands, 2010. Springer-Verlag.

[BE86] N Belkatir and J Estublier. Experience with a data base of programs. *SIGPLAN Not.*, 22(1):84–91, 1986.

[Beu08] D. Beuche. Modeling and building software product lines with pure::variants. In *Proceedings of the 2008 12th International Software Product Line Conference (SPLC '08)*, page 358, Washington, DC, USA, 2008. IEEE Computer Society.

[Bey11] D. Beyer. Crocopat. http://www.sosy-lab.org/ dbeyer/CrocoPat/, March 2011.

[BHJ+03] A. Birk, G. Heller, I. John, K. Schmid, T. von der Masen, and K. Muller. Product line engineering: The state of the practice. *IEEE Software*, 20(6):52–60, 2003.

[BKB+07] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software*, 80(4):571 – 583, 2007. Software Performance, 5th International Workshop on Software and Performance.

[BLS03] D. Batory, J. Liu, and J. N. Sarvela. Refinements and multidimensional separation of concerns. *SIGSOFT Software Engineering Notes*, 28(5):48–57, 2003.

[BM09] J. Barreiros and A. Moreira. Managing features and aspect interactions in software product lines. In *Proceedings of the 4th International Conference on Software Engineering Advances (ICSEA '09)*, pages 506–511, Porto, Portugal, 2009. IEEE Computer Society.

[Bre05] M. Breen. Experience of using a lightweight formal specification method for a commercial embedded system product line. *Requirements Engineering*, 10(2):161–172, 2005.

[BS10a] T. Berger and S. She. Formal semantics of the CDL language. `www.informatik.uni-leipzig.de/~berger/cdl_semantics.pdf`, 2010.

[BS10b] Inc BigLever Software. Biglever. http://www.biglever.com/index.html, May 2010.

[BSL+10] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Variability modeling in the real: a perspective from the operating systems domain. In *Proceedings of the 25th International Conference on Automated Software Engineering (ASE'10)*, pages 73–82, Antwerp, Belgium, 2010. ACM.

[BSR04] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling stepwise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[BSRC10] D. Benavides, S. Segura, and A. Ruiz-Cortes. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 35(6):615 – 636, 2010.

[BST10] C. Barrett, A. Stump, and C. Tinelli. The smt-lib standard: Version 2.0. `http://www.smtlib.org/`, 2010.

[BSTRC07] D. Benavides, D. Segura, P. Trinidad, and A. Ruiz Cortés. Fama: Tooling a framework for the automated analysis of feature models. In *Proceedings of the 1st International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'07)*, pages 129–134, Limerick, Ireland, 2007. University of Duisburg-Essen.

[BTA05] D. Benavides, P. Trinidad, and Ruiz-Cortez A. Automated reasoning on feature models. In *Proceedingss of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, pages 491–503, Porto, Portugal, 2005. Springer.

[BTN+08] G. Botterweck, S. Thiel, D. Nestor, S. bin Abid, and C. Cawley. Visual tool support for configuring and understanding software product lines. In *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pages 77–86, Limerick, Ireland, 2008. IEEE Computer Society.

[BTV09] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, pages 307–321, York, UK, 2009. Springer-Verlag.

[CAB09] L. Chen and M. Ali Babar. A survey of scalability aspects of variability modeling approaches. In *Workshop on Scalable Modeling Techniques for Software Product Lines (SCALE'09)*, pages 119–126, San Francisco, CA, USA, 2009.

[CAB11] L. Chen and M. Ali Babar. A systematic review of evaluation of variability management approaches in software product lines. *Information and Software Technology*, 53(4):344–362, 2011.

[CABA09] L. Chen, M. Ali Babar, and Nour Ali. Variability management in software product lines: A systematic review. In *Proceedings of the 13th Software Product Line Conference (SPLC'09)*, pages 81–90, San Francisco, CA, USA, 2009.

[CBH11] A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of tvl. *Science of Computer Programming*, In Press:–, 2011.

[CGR+12] K. Czarnecki, P. Grunbacher, R. Rabiser, K. Schmid, and A. Wasowski. Cool features and tough decisions: Two decades of variability modeling. In *Proceedings of the 6th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS'12)*, Leipzig, Germany, 2012. ACM Press.

[CHBT10] C. Cawley, P. Healy, G. Botterweck, and S. Thiel. Research tool to support feature configuration in software product lines. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10)*, pages 179–182. University of Duisburg-Essen, 2010.

[CHE04] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration using feature models. In *Proceedings of the 3rd International Software Product Line Conference (SPLC'04)*, pages 266–283, Boston, MA, USA, 2004. Springer.

[CHE05] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[CHH09a] A. Classen, A. Hubaux, and P. Heymans. Analysis of feature configuration workflows (poster). In *Proceedings of the 17th IEEE International Requirements Engineering Conference (RE'09)*, Atlanta, Georgia, USA, 2009.

[CHH09b] A. Classen, A. Hubaux, and P. Heymans. A formal semantics for multi-level staged configuration. In *Proceedings of the 3rd International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'09)*, pages 51–60, Sevilla, Spain, 2009.

[Cho11] Choco solver. http://www.emn.fr/z-info/choco-solver/, March 2011.

[CHS08] A. Classen, P. Heymans, and P.-Y. Schobbens. What's in a Feature: A Requirements Engineering Perspective. In José Luiz Fiadeiro and Paola Inverardi, editors, *Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE'08), held jointly with*

(ETAPS'08), volume 4961 of *LNCS*, pages 16–30. Springer, 2008.

[CHS+10] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *32nd International Conference on Software Engineering (ICSE'10)*, ICSE '10, pages 335–344, New York, NY, USA, 2010. ACM.

[CKK06] K. Czarnecki, C. H. P. Kim, and K. T. Kalleberg. Feature models are views on ontologies. In *Proceedings of the 10th International Software Product Line Conference (SPLC'06)*, pages 41–51, Maryland, USA, 2006. IEEE Computer Society.

[CLK08] H. Cho, K. Lee, and K. C. Kang. Feature relation and dependency management: An aspect-oriented approach. In *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pages 3–11, Limerick, Ireland, 2008. IEEE Computer Society.

[CLLK09] H. Choi, K. Lee, J. Lee, and K. C. Kang. Multiple views of feature models to manage complexity. In *Workshop on Scalable Modeling Techniques for Software Product Lines (SCALE 2009)*, pages 127–133, 2009.

[CN01] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Con07] Consultative Committee for Space Data Systems (CCSDS). *CCSDS File Delivery Protocol (CFDP): Blue Book, Issue 4*. Number CCSDS 727.0-B-4. National Aeronautics and Space Administration (NASA), January 2007.

[Con08] Four types of configurators: Which one is right for your business? Configure One, Inc., 2008. White Paper.

[Con11] Software Product Line Conference. Product line hall of fame. http://www.splc.net/fame.html, January 2011.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd annual ACM symposium on Theory of Computing (STOC '71)*, pages 151–158. ACM, 1971.

[Cor93] Software Productivity Consortium Services Corporation. Reuse-driven software processes guidebook. Technical Report SPC-92019-CMC, 1993. Version 02.00.03.

[CP06] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *5th International Conference on Generative programming and Component Engineering (GPCE'06)*, pages 211–220, New York, NY, USA, 2006. ACM.

[CRB04] A. Colyer, A. Rashid, and G. Blair. On the separation of concerns in program families. Technical Report COMP-001-2004, Lancaster University, 2004.

[CSW08] K. Czarnecki, S. She, and A. Wasowski. Sample spaces and feature models: There and back again. In *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pages 22–31, Limerick, Ireland, 2008. IEEE Computer Society.

[CTH08] Ciarán Cawley, Steffen Thiel, and Patrick Healy. Visualising variability relationships in software product lines. In *Proceedings of the 2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE'08) collocated with SPLC'08*, pages 329–333, Limerick, Ireland, 2008.

[CW98] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30:232–282, June 1998.

[CW07] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *Proceedings of the 11th International Software Product Line Conference(SPLC'07)*, pages 23–34, Kyoto, Japan, 2007. IEEE Computer Society.

[CZZM05] K. Chen, W. Zhang, H. Zhao, and H. Mei. An approach to constructing feature models based on requirements clustering. In *Proceedings of the 13th International Conference on Requirements Engineering (RE'05)*, pages 31–40, Paris, France, 2005. IEEE Computer Society.

[CZZM06] K. Chen, H. Zhao, W. Zhang, and H. Mei. Identification of crosscutting requirements based on feature dependency analysis. In *Proceedings of the 14th International Conference on Requirements Engineering (RE'06)*, pages 300–303, Minneapolis, MN, USA, 2006. IEEE Computer Society.

[Dar91] S. Dart. Concepts in configuration management systems. In *Proceedings of the 3rd international workshop on Software*

configuration management, pages 1–18, New York, NY, USA, 1991. ACM.

[Dar99] S. Dart. Content change management: Problems for web systems. In *SCM-9: Proceedings of the 9th International Symposium on System Configuration Management*, pages 1–16, London, UK, 1999. Springer-Verlag.

[DFdJV03] E. Dolstra, G. Florijn, M. de Jonge, and E. Visser. Capturing timeline variability with transparent configuration environments. In Jan Bosch and Peter Knauber, editors, *IEEE Workshop on Software Variability Management (SVM'03)*, Portland, Oregon, May 2003. IEEE.

[Dij82] E. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[DMB08] L. De Moura and N. Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems (TACAS/ETAPS'08)*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[DMH+07] G. Delannay, K. Mens, P. Heymans, P.-Y Schobbens, and J.-M. Zeippen. PloneGov as an Open Source Product Line. In *Proceeding of the International Workshop on Open Source Software and Product Lines (OSSPL'07), colocated with SPLC'07*, Kyoto, Japan, 2007. IEEE Computer Society.

[DPC+08] L. Demonceau, P. Parisis, M. Ciccone, G. Furano, and R. Blommestijn. CCSDS file delivery protocol for future ESA missions. In *Proceedings of DAta Systems In Aerospace (DASIA'08), International Space System Engineering Conference*, Palma de Majorca, Spain, 2008. ESA Publications.

[DR03] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th international conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'03)*, pages 78–95, Anaheim, California, USA, 2003. ACM.

[DRvdA+06] A. Dreiling, M. Rosemann, W. van der Aalst, L. Heuser, and K. Schulz. Model-based software configuration: patterns and languages. *European Journal of Information Systems*, 15:583–600, 2006.

[DS05] S. Deelstra and J. Sinnema, M.and Bosch. Product derivation in software product families: A case study. *Journal of Systems and Software*, 74(2):173–194, 2005.

[DSB09] S. Deelstra, M. Sinnema, and J. Bosch. Variability assessment in software product families. *Information Software Technology*, 51(1):195–218, 2009.

[DVALV10] R. Demeyer, M. Van Assche, L. Langevine, and W. Vanhoof. Declarative workflows to efficiently manage flexible and advanced business processes. In *Proceedings of the 12th international ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'10)*, pages 209–218, Hagenberg, Austria, 2010. ACM.

[DVC+07] B. Desmet, J. Vallejos, P. Costanza, W. De Meuter, and T. D'Hondt. *Modeling and Using Context*, chapter Context-Oriented Domain Analysis, pages 178–191. Springer Berlin / Heidelberg, 2007.

[eCo11] eCos. eCos User Guide. http://ecos.sourceware.org/docs-latest/user-guide/ecos-user-guide.html, March 2011.

[EDA97] J. Estublier, S. Dami, and M. Amiour. High level process modeling for scm systems. In Reidar Conradi, editor, *Software Configuration Management*, volume 1235 of *Lecture Notes in Computer Science*, pages 81–97. Springer Berlin / Heidelberg, 1997.

[ELF08] A. Egyed, E. Letier, and A. Finkelstein. Generating and evaluating choices for fixing inconsistencies in UML design models. In *Proceedings of the 23rd International Conference on Automated Software Engineering (ICSE'08)*, pages 99–108, Leipzig, Germany, 2008. IEEE Computer Society.

[ELSP08] C. Elsner, D. Lohmann, and W. Schröder-Preikschat. Towards separation of concerns in model transformation workflows. In *Workshop on Early Aspects (EA'08) collocated with SPLC'08*, pages 81–88, Limerick, Ireland, 2008.

[ELvdH+05] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *ACM Transaction on Software Engineering Methodology*, 14(4):383–430, 2005.

[EM08] L. Etxeberria and G. S. Mendieta. Variability driven quality evaluation in software product lines. In *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pages 243–252, Limerick, Ireland, 2008. IEEE Computer Society.

[EN96] S. M. Easterbrook and B. A. Nuseibeh. Using viewpoints for inconsistency management. *Software Engineering Journal*, 11(1), 1996.

[EN06] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems (5th Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[ES11] N. Eén and N. Sörensson. Minisat. http://minisat.se/, March 2011.

[Est00] J. Estublier. Software configuration management: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*, pages 279–289, Limerick, Ireland, 2000. ACM.

[Fau01] S. R. Faulk. Product-line requirements specification (prs): An approach and case study. In *Proceedings of the 5th International Conference on Requirements Engineering (RE'01)*, pages 48–55, Toronto, ON, Canada, 2001. IEEE Computer Society.

[Fei91] P. H. Feiler. Configuration management models in commercial environments. Technical Report CMU/SEI-91-TR-7, Software Engineerning Institute, Carnegie Mellon University, Pittsburgh, PA, USA, March 1991.

[FFB02] D. Fey, R. Fajta, and A. Boros. Feature modeling: A meta-model to enhance usability and usefulness. In *Proceedings of the 2nd International Software Product Line Conference (SPLC'02)*, pages 198–216, London, UK, 2002. Springer-Verlag.

[FFJS04] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence*, 152:213–234, February 2004.

[FFS+09] A. Felfernig, G. Friedrich, M. Schubert, M. Mandl, M. Mairitsch, and E. Teppan. Plausible repairs for inconsistent requirements. In *Proceedings of the 21st international joint conference on Artifical intelligence (IJCAI'09)*, pages 791–796,

San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[GBPLM04] B. Gonzales-Baixauli, J.C.S. Prado Leite, and J. Mylopoulos. Visual variability analysis for goal models. In *Proceedings of the 12th International Conference on Requirements Engineering (RE'04)*, pages 198–207, Kyoto, Japan, 2004. IEEE.

[GCB+10] C. Gauthier, A. Classen, Q. Boucher, P. Heymans, M-A. D. Storey, and M. Mendonça. Xtof - a tool for tag-based product line implementation. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'10)*, pages 163–166, Linz, Austria, 2010. Universität Duisburg-Essen.

[GG96] B. Gulla and J. Gorman. Experiences with the use of a configuration language. In *Proceedings of the SCM-6 Workshop on System Configuration Management (ICSE'96)*, pages 198–219, London, UK, 1996. Springer-Verlag.

[GGJZ00] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, 2000.

[Gil10] P. Gilbert. The Next Decade of BPM. http://vimeo.com/15680641, Septembre 2010. Keynote presentation at the 8th International Conference on Business Process Management (BPM'10).

[Git11] Git. Git: The fast version control system. http://git-scm.com/, October 2011.

[GK99] A. Günter and C. Kühn. Knowledge-based configuration: Survey and future directions. In *Proceedings of the 5th Biannual German Conference on Knowledge-Based Systems (XPS'99)*, pages 47–66, London, UK, 1999. Springer-Verlag.

[GRDL09] P. Grünbacher, R. Rabiser, D. Dhungana, and M. Lehofer. Structuring the product line modeling space: Strategies and examples. In *Proceedings of the 3rd International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'09)*, pages 77–82, Sevilla, Spain, 2009.

[GS08] H. Gomaa and M. E. Shin. Multiple-view modelling and metamodelling of software product lines. *IET Software*, 2(2):94–122, 2008.

[GSW89] R. Greiner, B. A. Smith, and R. W. Wilkerson. A correction to the algorithm in reiter's theory of diagnosis. *Artificial Intelligence*, 41:79–88, November 1989.

[GvdAJVLR07] F. Gottschalk, W. M.P. van der Aalst, M. H. Jansen-Vullers, and M. La Rosa. Configurable workflow models. *International Journal of Cooperative Information Systems*, 17(2):177–221, 2007.

[Har06] U. Harlou. *Developing product families based on architectures : Contribution to a theory of product families*. PhD thesis, Technical University of Denmark, Department of Mechanical Engineering, Engineering Design and Product Development, 2006.

[HBH+10] A. Hubaux, Q. Boucher, H. Hartmann, R. Michel, and P. Heymans. Evaluating a textual feature modelling language: Four industrial case studies. In *Proceedings of the 3rd International Conference on Software Language Engineering (SLE'10)*, volume 6563, pages 337–356, Eindhoven, The Netherlands, 2010. Springer Berlin / Heidelberg.

[HCH09] A. Hubaux, A. Classen, and P. Heymans. Formal modelling of feature configuration workflow. In *Proceedings of the 13th International Software Product Lines Conference (SPLC'09)*, pages 221–230, San Francisco, CA, USA, 2009. ACM Press.

[HCMH10] A. Hubaux, A. Classen, M. Mendonça, and P. Heymans. A preliminary review on the application of feature diagrams in practice. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10)*, pages 53–59, Linz, Austria, January 2010.

[HHB08] A. Hubaux, P. Heymans, and D. Benavides. Variability modelling challenges from the trenches of an open source product line re-engineering project. In *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pages 55–64, Limerick, Ireland, 2008. IEEE Computer Society.

[HHSD10] A. Hubaux, P. Heymans, P.-Y. Schobbens, and D. Deridder. Towards multi-view feature-based configuration. In *Proceedings of the 16th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ'10)*, pages 106–112, Essen, Germany, 2010. Springer-Verlag.

[HK07] I. Habli and T. Kelly. Challenges of establishing a software product line for an aerospace engine monitoring system. In *Proceedings of the 11th Software Product Line Conference (SPLC'07)*, pages 193–202, Kyoto, Japan, 2007. IEEE Computer Society.

[HMR08] L. Hvam, N. Henrik Mortensen, and J. Riis. *Product Customization*. Springer-Verlag Berlin Heidelberg, 2008.

[HP03] G. Halmans and K. Pohl. Communicating the variability of a software-product family to customers. *Software and Systems Modeling*, 2:15–36, 2003. 10.1007/s10270-003-0019-9.

[HR00] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Technical report, Weizmann Institute Of Sience, Jerusalem, Israel, Israel, 2000.

[HSJ+04] T. Hadzic, S. Subbarayan, R. M. Jensen, H. R. Andersen, J. Møller, and H. Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. In *PETO Conference*, pages 131–138. DTU-tryk, June 2004.

[HSS+10] F. Heidenreich, P. Sánchez, J. Santos, S. Zschaler, M. Alférez, J. Araújo, L. Fuentes, U. Kulesza, A. Moreira, and A. Rashid. Relating feature models to other models of a software product line - a comparative study of featuremapper and vml*. *Theory of Aspect-Oriented Software Development*, 7:69–114, 2010.

[HSSF06] S. O. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. Using product line techniques to build adaptive systems. In *Proceedings of the 10th International Software Product Line Conference (SPLC'06)*, pages 141–150, Maryland, USA, 2006. IEEE Computer Society.

[HT08] H. Hartmann and T. Trew. Using feature diagrams with context variability to model multiple product lines for software supply chains. In *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pages 12–21, Limerick, Ireland, 2008. IEEE Computer Society.

[HTH11] A. Hubaux, T. T. Tun, and P. Heymans. Separation of concerns in feature diagram languages: A systematic survey. *ACM Computing Surveys*, 2011. Under review.

[HXC11] A. Hubaux, Y. Xiong, and K. Czarnecki. Configuration challenges in linux and ecos: A survey. Technical Report

GSDLAB-TR 2011-09-29, Generative Software Development Laboratory, University of Waterloo, 2011.

[HXC12] A. Hubaux, Y. Xiong, and K. Czarnecki. A survey of configuration challenges in linux and ecos. In *Proceedings of the Sixth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'12)*, Leipzig, Germany, 2012. ACM Press.

[IBM11] IBM. Ibm rational jazz technology platform. http://www-01.ibm.com/software/rational/jazz/, October 2011.

[IEE] IEEE Std. No. 610.12-1990, Glossary of Software Engineering Terminology.

[Jac95] M. Jackson. *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*. ACM Press, 1995.

[Jac11] JaCoP. http://jacop.osolpro.com/, March 2011.

[Jan10] M. Janota. *SAT Solving in Interactive Configuration*. PhD thesis, University College Dublin, 2010.

[JBGMS10] M. Janota, G. Botterweck, R. Grigore, and J. Marques-Silva. How to complete an interactive configuration process? In *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'10)*, pages 528–539, Czech Republic, 2010. Springer-Verlag.

[JL06] D. Jannach and J. Liegl. Conflict-directed relaxation of constraints in content-based recommender systems. In Moonis Ali and Richard Dapoigny, editors, *Advances in Applied Artificial Intelligence*, volume 4031 of *Lecture Notes in Computer Science*, pages 819–829. Springer Berlin / Heidelberg, 2006.

[JM11] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'11)*, San Jose, California, USA, pages 437–446. ACM, 2011.

[Jun04] U. Junker. Quickxplain: preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th conference on Artifical intelligence (AAAI'04)*, pages 167–172. AAAI Press, 2004.

[Jun06] U. Junker. *Handbook of Constraint Programming*, chapter Configuration, pages 837–873. Foundations of Artificial Intelligence. Elsevier North-Holland, Inc., 2006.

[KÏ0] C. Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0.* PhD thesis, Otto-von-Guericke-Universität Magdeburg, Germany, 2010.

[KÏ1] C. Kästner. Cide: Virtual separation of concerns. http://www.fosd.de/cide, March 2011.

[KAK08] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 311–320, New York, NY, USA, 2008. ACM.

[KCH+90] K. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Software Engineering Institute, Carnegie Mellon University, 1990.

[KDK+02] K. C. Kang, P. Donohoe, Eunman Koh, Jaejoon Lee, and Kwanwoo Lee. Using a marketing and product plan as a key driver for product line asset development. In *Proceedings of the 2nd International Software Product Line Conference (SPLC'02)*, pages 366–382, San Diego, CA, USA, 2002. Springer.

[KHC05] S. D. Kim, J. S. Her, and S. H. Chang. A theoretical foundation of variability in component-based development. *Information and Software Technology*, 47(10):663–673, 2005.

[Kit04] B. A. Kitchenham. Procedures for undertaking systematic reviews. Technical Report 0400011T.1, Computer Science Department, Keele University (TR/SE-0401) and National ICT Australia Ltd, 2004.

[KKL+98] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998. 10.1023/A:1018980625587.

[Kru02] C. W. Krueger. Variation management for software production lines. In *Proceedings of the 2nd International Software Product Line Conference (SPLC'02)*, pages 37–48, San Diego, CA, USA, 2002. Springer.

[Kru07] C. W. Krueger. Biglever software gears and the 3-tiered spl methodology. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented Programming Systems and Applications Companion (OOPSLA'07)*, pages 844–845, New York, NY, USA, 2007. ACM.

[KSG06] M. Kircher, C. Schwanninger, and I. Groher. Transitioning to a software product family approach - challenges and best practices. In *Proceedings of the 10th International Software Product Line Conference (SPLC'06)*, pages 163–171, Maryland, USA, 2006. IEEE Computer Society.

[KTS+09] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: A tool framework for feature-oriented software development. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 311–320, Vancouver, ON, Canada, 2009. IEEE.

[LB11] D. Le Berre. Sat4j: Bringing the power of sat technology to the java platform. http://www.sat4j.org/, March 2011.

[LK06] J. Lee and K. C. Kang. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In *Proceedings of the 10th International Software Product Line Conference (SPLC'06)*, pages 131–140, Baltimore, Maryland, USA, 2006. IEEE Computer Society.

[LKK04] J. Lee, K. C. Kang, and S. Kim. A feature-based approach to product line production planning. In *Proceedings of the 3rd International Software Product Line Conference (SPLC'04)*, pages 183–196, Boston, MA, USA, 2004. Springer.

[LKKP06] K. Lee, K. C. Kang, M. Kim, and S. Park. Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In *Proceedings of the 10th International Software Product Line Conference (SPLC'06)*, pages 103–112, Baltimore, Maryland, USA, 2006. IEEE Computer Society.

[LKL02] K. Lee, K. C. Kang, and J. Lee. Concepts and guidelines of feature modeling for product line software engineering. In *Proceedings of the 7th International Conference on Software Reuse*, pages 62–77, London, UK, 2002. Springer-Verlag.

[LLT05] J. J. Liu, R. R. Lutz, and J. M. Thompson. Mapping concern space to software architecture: a connector-based approach. In *Proceedings of the Workshop on Modeling and Analysis of Concerns in Software (MACS'05)*, pages 1–5, New York, NY, USA, 2005. ACM.

[LM08] J. Lang and P. Marquis. On propositional definability. *Artificial Intelligence*, 172(8-9):991–1017, 2008.

[LP08] K. Lauenroth and K. Pohl. Dynamic consistency checking of domain requirements in product line engineering. In *Proceedings of the 16th International Requirements Engineering Conference (RE'08)*, pages 193–202, Catalunya, Spain, 2008. IEEE Computer Society.

[LRvdADTH08] M. La Rosa, W. van der Aalst, M. Dumas, and A. Ter Hofstede. Questionnaire-based variability modeling for system configuration. *Software and Systems Modeling*, 8(2):251–274, 2008.

[Lut08] R. R. Lutz. Enabling verifiable conformance for product lines. In *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pages 35–44, Limerick, Ireland, 2008. IEEE Computer Society.

[Man02] M. Mannion. Using First-Order Logic for Product Line Model Validation. In *Proceedings of the 2nd Software Product Line Conference (SPLC'02)*, pages 176–187, San Diego, CA, USA, 2002. Springer.

[MBC08] M. Mendonça, T. Tonelli Bartolomei, and D. Cowan. Decision-making coordination in collaborative product configuration. In *ACM symposium on Applied computing (SAC'08)*, pages 108–113, New York, NY, USA, 2008. ACM.

[MBC09] M. Mendonça, M. Branco, and D. Cowan. S.p.l.o.t.: software product lines online tools. In *Companion of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'09)*, pages 761–762, Orlando, Florida, USA, 2009. ACM.

[MCHB11] R. Michel, A. Classen, A. Hubaux, and Q. Boucher. A formal semantics for feature cardinalities in feature diagrams. In *Proceedings of the 5th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'11)*, pages 82–89, Namur, Belgium, 2011. ACM Press.

[MCMdO08] M. Mendonça, D. D. Cowan, W. Malyk, and T. C. de Oliveira. Collaborative product configuration: Formalization and efficient algorithms for dependency analysis. *Journal of Software*, 3(2):69–82, 2008.

[Men09] M. Mendonça. *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, University of Waterloo, 2009.

[Men10] M. Mendonça. Splot. http://www.splot-research.org/, May 2010.

[MHP+07] A. Metzger, P. Heymans, K. Pohl, P.-Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Proceedings of 15th International Conference on Requirements Engineering (RE'07)*, pages 243–253, Delhi, India, 2007. IEEE Computer Society.

[ML88] A. Mahler and A. Lampen. "shape - a software configuration management tool". In B. G. Teubner, editor, *Proceedings of the International Workshop on Software Version and Configuration Control*, pages 228–243, Grassau, West-Germany, 1988.

[Moe11] R. F. Moeller. Racer - Renamed Abox and Concept Expression Reasoner. http://www.sts.tu-harburg.de/ r.f.moeller/racer/, March 2011.

[MRA05] A. Moreira, Awais Rashid, and João Araújo. Multi-dimensional separation of concerns in requirements engineering. In *13th International Conference on Requirements Engineering (RE'05)*, pages 285–296, 2005.

[MSA09] M. Mannion, J. Savolainen, and T. Asikainen. Viewpoint-oriented variability modeling. *Computer Software and Applications Conference, Annual International*, 1:67–72, 2009.

[MSDS10] S. Mani, V. S. Sinha, P. Dhoolia, and S. Sinha. Automated support for repairing input-model faults. In *Proceedings of the 25th international conference on Automated Software Engineering (ASE'10)*, pages 195–204, Antwerp, Belgium, 2010. ACM.

[MWC09] M. Mendonça, A. Wąsowski, and K. Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference (SPLC'09)*,

pages 231–240, San Francisco, CA, USA, 2009. Carnegie Mellon University.

[NE08] N. Niu and S. M. Easterbrook. Extracting and modeling product line functional requirements. In *16th International Requirements Engineering Conference (RE'08)*, pages 155–164, Barcelona, Spain, 2008. IEEE Computer Society.

[NEF03] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 455–464, Portland, Oregon, 2003. IEEE Computer Society.

[NI07] E. Niemelä and A. Immonen. Capturing quality requirements of product family architecture. *Information Software Technology*, 49(11-12):1107–1120, 2007.

[NK08] N. Noda and T. Kishi. Aspect-oriented modeling for variability management. In *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pages 213–222, Limerick, Ireland, 2008. IEEE Computer Society.

[NOST07] D. Nestor, L. O'Malley, E. Sikora, and S. Thiel. Visualisation of variability in software product line engineering. In *Proceedings of the 1st International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'07)*, pages 1–8, Limerick, Ireland, 2007. University of Duisburg-Essen.

[NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53:937–977, November 2006.

[OJ09] A. Olszak and B. N. Jørgensen. Remodularizing java programs for comprehension of features. In *Proceedings of the First International Workshop on Feature-Oriented Software Development (FOSD'09)*, pages 19–26, Denver, Colorado, 2009. ACM Press.

[Olh03] J. Olhager. Strategic positioning of the order penetration point. *International Journal of Production Economics*, 85(3):319–329, 2003. Structuring and Planning Operations.

[OPFP07] B. O'Sullivan, A. Papadopoulos, B. Faltings, and P. Pu. Representative explanations for over-constrained problems. In

Proceedings of the 22nd conference on Artificial intelligence (AAAI'07), pages 323–328. AAAI Press, 2007.

[ORRT09] P. O'Leary, R. Rabiser, I. Richardson, and S. Thiel. Important issues and key activities in product derivation: Experiences from two independent research projects. In *Proceedings of the 13th Software Product Line Conference (SPLC'09)*, pages 121–130, San Francisco, CA, USA, 2009. ACM Press.

[OWL11] Owl. http://www.w3.org/TR/owl-features/, March 2011.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communication of the ACM*, 15(12):1053–1058, 1972.

[Par76] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[PBD10] A. Pleuss, G. Botterweck, and D. Dhungana. Integrating Automated Product Derivation and Individual User Interface Design. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'10)*, pages 69–76, Linz, Austria, 2010. University of Duisburg-Essen.

[PBvdL05] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[PCCW93] M.C. Paulk, B. Curtis, M.B. Chrissis, and C.V. Weber. Capability maturity model, version 1.1. *Software, IEEE*, 10(4):18–27, July 1993.

[PNX+11] L. Passos, M. Novakovic, Y. Xiong, T. Berger, K. Czarnecki, and A. Wasowski. A study of non-boolean constraints in variability models of an embedded operating system. In *Proceedings of the 3rd International Workshop on Feature-Oriented Software Development (FOSD'11)*, pages 1–8, Munich, Germany, 2011. ACM.

[Pre04] R. S. Pressman. *Software engineering: a practitioner's approach (7th ed.)*, chapter Change Management, pages 771–800. McGraw-Hill, Inc., New York, NY, USA, 2004.

[psG06] pure-systems GmbH. Variant management with pure::variants. http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf, 2006. Technical White Paper.

[Rei87] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, April 1987.

[Rei09] M.-O. Reiser. Core concepts of the compositional variability management framework (cvm). Technical report, Technische Universität Berlin, 2009.

[RGD07] R. Rabiser, P. Grunbacher, and D. Dhungana. Supporting product derivation by adapting and augmenting variability models. In *Proceedings of the 11th International Software Product Line Conference (SPLC07)*, pages 141–150, Sept. 2007.

[RPTB10] M. Ribeiro, H. Pacheco, L. Teixeira, and P. Borba. Emergent feature modularization. In *Proceedings of the International Conference Companion on Object-Oriented Programming Systems Languages and Applications (OOPSLA'10)*, SPLASH '10, pages 11–18, Reno/Tahoe Nevada, USA, 2010. ACM Press.

[RW06] M.-O. Reiser and M. Weber. Managing highly complex product families with multi-level feature trees. In *14th International Conference on Requirements Engineering (RE'06)*, pages 146–155, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[RW07] M.-O. Reiser and M. Weber. Multi-level feature trees. *Requirements Engineering*, 12(2):57–75, 2007.

[SCA09] *Workshop on Scalable Modeling Techniques for Software Product Lines (SCALE'09) at SPLC'09, San Francisco, USA*, 2009.

[SG05] M. Saleh and H. Gomaa. Separation of concerns in software product line engineering. In *Workshop on Modeling and Analysis of Concerns in Software (WACS'05)*, pages 1–5, New York, NY, USA, 2005. ACM.

[SHTB06] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *Proceedings of the 14th IEEE International Requirements*

Engineering Conference (RE'06), pages 139–148, Minneapolis, Minnesota, USA, September 2006.

[SHTB07] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.

[Sin05] C. Sinz. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, pages 827–831, Barcelona, Spain, 2005. Springer.

[SK01] J. Savolainen and J. Kuusela. Consistency management of product line requirements. In *Proceedings of the 5th International Conference on Requirements Engineering (RE'01)*, pages 40–47, Toronto, ON, Canada, 2001. IEEE Computer Society.

[SK09] K. Schmid and C. Kroher. An analysis of existing software configuration systems. In *Proceedings of the Workshop on Dynamic Software Product Lines (DSPL'09)*, pages 2–7, San Francisco, CA, USA, 2009.

[SLB+11] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 461–470, Waikiki, Honolulu, Hawaii, 2011. ACM Press.

[SRBS04] J. Scheffczyk, P. Rödig, U. M. Borghoff, and L. Schmitz. Managing inconsistent repositories via prioritized repairs. In *Proceedings of the 2004 ACM symposium on Document engineering (DocEng '04)*, pages 137–146, Milwaukee, Wisconsin, USA, 2004. ACM.

[SRG11] K. Schmid, R. Rabiser, and P. Grunbacher. A comparison of decision modeling approaches in product lines. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS'11)*, pages 119–126, Namur, Belgium, 2011. ACM.

[SSS07] F. Shull, J. Singer, and D. I. K. Sjøberg. *Guide to Advanced Empirical Software Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, 1977.

[SvGB05] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software Practice and Experience*, 35(8):705–754, 2005.

[SZ01] G. Spanoudakis and A. Zisman. Inconsistency management in software engineering: Survey and open research issues. In K Chang S., editor, *Handbook of Software Engineering and Knowledge Engineering*, pages 329–380. World Scientific Publishing Co, 2001.

[TBC+09] T. T. Tun, Q. Boucher, A. Classen, A. Hubaux, and P. Heymans. Relating requirements and feature configurations: A systematic approach. In *Proceedings of the 13th International Software Product Lines Conference (SPLC'09)*, pages 201–210, San Francisco, CA, USA, 2009. ACM Press.

[TBD+08] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81(6):883–896, 2008.

[TH02] S. Thiel and A. Hein. Systematic integration of variability into product line architecture design. In *Proceedings of the 2nd International Software Product Line Conference (SPLC'02)*, pages 130–153, San Diego, CA, USA, 2002. Springer.

[TH03] J. M. Thompson and M. P. E. Heimdahl. Structuring product family requirements for n-dimensional and hierarchical product lines. *Requirements Engineering*, 8(1):42–54, 2003.

[Tho54] H. D. Thoreau. *Walden*, chapter Conclusion. Everyman's Library. 1854.

[tHvdAAR09] A. ter Hofstede, W. van der Aalst, M. Adams, and N. Russell. *Modern Business Process Automation: YAWL and its Support Environment*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[TOHS99] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Jr. Sutton. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 107–119, Los Angeles, CA, USA, 1999. ACM Press.

[Tri08] J.-C. Trigaux. *Quality of Feature Diagram Languages: Formal Evaluation and Comparison.* PhD thesis, University of Namur, Faculty of Computer Science, September 2008.

[Tsa93] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press Limited, 1993.

[TSMS96] J. Tiihonen, T. Soininen, T. Männistö, and R. Sulonen. State-of-the-practice in product configuration - a survey of 10 cases in the finnish industry. In *In Knowledge Intensive CAD*, pages 95–114. Chapman & Hall, 1996.

[UC04] S. Uchitel and M. Chechik. Merging partial behavioural models. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE'04)*, pages 43–52, CA, USA, Newport Beach, 2004. ACM Press.

[UDH09] H. Unphon, Y. Dittrich, and A. Hubaux. Taking care of cooperation when evolving socially embedded systems: The plonemeeting case. In *Proceedings of the Workshop on Cooperative and Human Aspects of Software Engineering (CHASE'09), collocated with ICSE'09*, pages 96–103, Vancouver, BC, Canada, 2009. IEEE Computer Society.

[VD01] B. Veer and J. Dallaway. The eCos component writer's guide. `ecos.sourceware.org/ecos/docs-latest/cdl-guide/cdl-guide.html`, 2001.

[vdAADTH04] W. van der Aalst, L. Aldred, M. Dumas, and A. Ter Hofstede. Design and Implementation of the YAWL System. In *Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE'04)*, pages 142–159, Riga, Latvia, 2004. Springer.

[vdAtH05] W. van der Aalst and A. ter Hofstede. Yawl: yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[vdAtHKB03] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.

[vdH04] A. van der Hoek. Design-time product line architectures for any-time variability. *Science of Computer Programming*, 53(3):285–304, 2004.

[vDK02] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10:1–17, 2002.

[vdLSR07] F. van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[VGBS01] J. Van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 45, Washington, DC, USA, 2001. IEEE Computer Society.

[vZ02] J. van Zyl. Product line architecture and the separation of concerns. In *Proceedings of the 2nd International Software Product Line Conference (SPLC'02)*, pages 90–109, San Diego, CA, USA, 2002. Springer.

[WBDS09] J. White, D. Benavides, B. Dougherty, and D. C. Schmidt. Automated reasoning for multi-step software product-line configuration problems. In *Proceedings of the 13th International Software Product Lines Conference (SPLC'09)*, pages 11–20, San Francisco, CA, USA, 2009. ACM Press.

[WD06] E. Wohlstadter and P. Devanbu. Aspect-oriented development of crosscutting features in distributed, heterogeneous systems. In A. Rashid and M. Aksit, editors, *Transactions on Aspect-Oriented Software Development II*, volume 4242 of *Lecture Notes in Computer Science*, pages 69–100. Springer Berlin / Heidelberg, 2006.

[Wha11] J. Whaley. Javabdd. http://javabdd.sourceforge.net/, March 2011.

[WJ09] B. Waldmann and P. Jones. Feature-oriented requirements satisfy needs for reuse and systems view. In *Proceedings of the 17th International Conference on Requirements Engineering (RE'09)*, pages 329–334, Atlanta, Georgia, USA, 2009. IEEE Computer Society.

[WJR91] J. P. Womack, D. T. Jones, and D. Roos. *The Machine That Changed the World : The Story of Lean Production*. Harper Perennial, October 1991.

[WLS+07] H. H. Wang, Y. F. Li, J. Sun, H. Zhang, and J. Pan. Verifying feature models using owl. *Web Semantics*, 5:117–129, June 2007.

[WSB+08] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated diagnosis of product-line configuration errors in feature models. In *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pages 225–234, Limercick, Ireland, 2008. IEEE Computer Society.

[WVvdA+09] M. T. Wynn, H.M.W. Verbeek, W. M. van der Aalst, A. H. ter Hofstede, and David Edmond. Business process verification - finally a reality. *Business Process Management Journal*, 15(1):74–92, 2009.

[WXH+10] B. Wang, Y. Xiong, Z. Hu, H. Zhao, W. Zhang, and H. Mei. A dynamic-priority based approach to fixing inconsistent feature models. In *Proceedings of the 13th international conference on Model driven engineering languages and systems (MODELS'10)*, pages 181–195, Oslo, Norway, 2010. Springer-Verlag.

[Wyn06] M. T. Wynn. *Semantics, Verification, and Implementation of Workflows with Cancellation Regions and OR-joins*. PhD thesis, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, November 2006.

[XHZ+09] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei. Supporting automatic model inconsistency fixing. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the symposium on The foundations of software engineering (ESEC/FSE'09)*, pages 315–324, Amsterdam, The Netherlands, 2009. ACM.

[Xio11] Y. Xiong. Configurator semantics of the cdl language. Technical Report GSDLAB-TR 2011-06-05, Generative Software Development Laboratory, University of Waterloo, 2011.

[ZHJ04] T. Ziadi, L. Helouet, and J.-M. Jezequel. Towards a uml profile for software product lines. In Frank van der Linden, editor, *Software Product-Family Engineering*, volume 3014 of *Lecture Notes in Computer Science*, pages 129–139. Springer Berlin / Heidelberg, 2004.

[ZJ97] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(1):1–30, 1997.

[ZMZ06] W. Zhang, H. Mei, and H. Zhao. Feature-driven requirement dependency analysis and high-level software design. *Requirements Engineering*, 11(3):205–220, 2006.

[ZZM08] H. Zhao, W. Zhang, and H. Mei. Multi-view based customization of feature models. *Journal of Frontiers of Computer Science and Technology*, 2(3):260–273, 2008.